REFERENCES

REFERENCES

ARE A C++ FEATURE THAT DRAMATICALLY
SIMPLIFY USING POINTERS

YOU DON'T NEED TO DEREFERENCE A REFERENCE :-)

NO NEED TO `malloc/free` OR `new/delete` THEM

REFERENCES CAN NEVER BE NULL

REFERENCES

```
int x = 5;
```

A PERFECTLY USUAL VARIABLE OF TYPE `int`

REFERENCES

```
int x = 5;
int& y = x;
```

AN 'INT REFERENCE' OR 'A REFERENCE TO AN INT': DECLARED TO BE OF TYPE `int&`

REFERENCES

```
int x = 5;
int& y = x;
```

AN 'INT REFERENCE' OR 'A REFERENCE TO AN INT': DECLARED TO BE OF TYPE `int&`

REFERENCES

```
int x = 5;
int& y = x;
```

NOW, WHEN YOU CHANGE `y`, YOU WILL CHANGE `x`!

REFERENCES

```
int x = 5;
int& y = x;
```

USE AN `int&` EXACTLY AS YOU WOULD AN `int`

BUT BE AWARE THAT IT IS PASSED-BY-REFERENCE (LIKE A POINTER)

REFERENCES ARE IN FACT, POINTERS - C++ JUST MAKES THEIR SYNTAX FAR EASIER TO USE

REFERENCES

REFERENCES ARE IN FACT, POINTERS - C++ JUST MAKES THEIR SYNTAX FAR EASIER TO USE

NO MEMORY ALLOCATION, MEMORY LEAKS, OR EVEN NULLS (A REFERENCE CAN NEVER BE NULL!)

REFERENCES

RULE #1: A REFERENCE MUST ALWAYS BE INITIALISED

```
int y = 3;
```

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

X

~~`int& x;`~~

~~`x = y;`~~

REFERENCES

RULE #2: REFERENCE REASSIGNMENTS DON'T DO QUITE WHAT YOU'D EXPECT THEM TO

REFERENCES

RULE #3: MULTIPLE REFERENCES TO THE SAME VALUE CAN EXIST - IF ONE IS MODIFIED, ALL GET MODIFIED

REFERENCES

RULE #4: REFERENCES CAN NEVER BE NULL

REFERENCES

RULE #5: REFERENCES CAN EXIST TO ANY TYPE (INCLUDING POINTERS)

REFERENCES

RULE #6:

C++ Standard 8.3.2/4:

> There shall be no references to references, **no arrays of references**, and no pointers to references.

REFERENCES

REFERENCES ARE GREAT FOR PASSING VALUES INTO FUNCTIONS

REFERENCES ARE SOMETIMES TRICKY FOR TAKING VALUES OUT OF FUNCTIONS

EXAMPLE 19:

DEFINE A VARIABLE OF TYPE REFERENCE, AND SEE HOW IT WORKS

EXAMPLE 19: DEFINE A VARIABLE OF TYPE REFERENCE, AND SEE HOW IT WORKS

```
int x = 5;
int& y = x;

cout << "Initially, x = " << x << " y = " << y << endl;
y = 10;
cout << "Change the value of y to " << y << endl;
cout << "Finally, x = " << x << " y = " << y << endl;
```

EXAMPLE 19: DEFINE A VARIABLE OF TYPE REFERENCE, AND SEE HOW IT WORKS

```
int x = 5;
int& y = x;

cout << "Initially, x = " << x << " y = " << y << endl;
y = 10;
cout << "Change the value of y to " << y << endl;
cout << "Finally, x = " << x << " y = " << y << endl;
```

DEFINE A SIMPLE VARIABLE

EXAMPLE 19: DEFINE A VARIABLE OF TYPE REFERENCE, AND SEE HOW IT WORKS

```
int x = 5;
int& y = x;
cout << "Initially, x = " << x << " y = " << y << endl;
y = 10;
cout << "Change the value of y to " << y << endl;
cout << "Finally, x = " << x << " y = " << y << endl;
```

DEFINE A REFERENCE AND SET IT TO THAT SAME VARIABLE

EXAMPLE 19: DEFINE A VARIABLE OF TYPE REFERENCE, AND SEE HOW IT WORKS

```
int x = 5;
int& y = x;

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

y = 10;

cout << "Change the value of y to " << y << endl;
cout << "Finally, x = " << x << " y = " << y << endl;
```

NOW MODIFY THE VALUE OF THE REFERENCE (NOT THE ORIGINAL!)

EXAMPLE 19: DEFINE A VARIABLE OF TYPE REFERENCE, AND SEE HOW IT WORKS

```
int x = 5;
int& y = x;

cout << "Initially, x = " << x << " y = " << y << endl;
y = 10;
cout << "Change the value of y to " << y << endl;
cout << "Finally, x = " << x << " y = " << y << endl;
```

TEST WHETHER THE VALUE OF THE ORIGINAL HAS CHANGED?

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Initially, x = 5 y = 5
Change the value of y to 10
Finally, x = 10 y = 10
```

YES IT HAS!

EXAMPLE 20:

UNDERSTAND HOW CONST REFERENCES WORK

EXAMPLE 20: UNDERSTAND HOW CONST REFERENCES WORK

```
int x = 5;
```

A PERFECTLY USUAL VARIABLE OF TYPE `int`

EXAMPLE 20: UNDERSTAND HOW CONST REFERENCES WORK

WON'T COMPILE!!!

```
int x = 5;
int& y = 5;
```

AN 'INT REFERENCE' OR 'A REFERENCE TO AN INT': DECLARED TO OF TYPE `int&`

EXAMPLE 20: UNDERSTAND HOW CONST REFERENCES WORK

WORKS!

```
int x = 5;
const int& y = 5;
```

TO HOLD A CONSTANT VALUE, OUR VARIABLE MUST BE DECLARED TO OF TYPE `const int&`

EXAMPLE 20: UNDERSTAND HOW CONST REFERENCES WORK

MUCH MORE ON CONST IN A BIT, FOR NOW JUST REMEMBER THIS LITTLE RULE :-)

EXAMPLE 21:

COMPARE THE OLD (C-STYLE) AND NEW (C++ REFERENCE) STYLE OF SWAPPING TWO VARIABLES

EXAMPLE 21: COMPARE THE OLD (C-STYLE) AND NEW (C++ REFERENCE) STYLE OF SWAPPING TWO VARIABLES

```
// writing the function
double swap(int *a,int *b)
{
   int temp = *a;
   a = b;
   *b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(&a,&b);
printf("a = %d, b = %d\n",a,b);
```

INCREDIBLY COMPLICATED CODE, FOR A REALLY SIMPLE OPERATION!

EXAMPLE 21: COMPARE THE OLD (C-STYLE) AND NEW (C++ REFERENCE) STYLE OF SWAPPING TWO VARIABLES

INCREDIBLY COMPLICATED CODE, FOR A REALLY SIMPLE OPERATION!

```
// writing the function
double swap(int *a,int *b)
{
   int temp = *a;
   a = b;
   *b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(&a,&b);
printf("a = %d, b = %d\n",a,b);
```

THE FUNCTION BODY MUST TAKE IN AND DEREFERENCE POINTERS

EXAMPLE 21: COMPARE THE OLD (C-STYLE) AND NEW (C++ REFERENCE) STYLE OF SWAPPING TWO VARIABLES

INCREDIBLY COMPLICATED CODE, FOR A REALLY SIMPLE OPERATION!

THE FUNCTION BODY MUST TAKE IN AND DEREFERENCE POINTERS

```
// writing the function
double swap(int *a,int *b)
{
   int temp = *a;
   a = b;
   *b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(&a,&b);
printf("a = %d, b = %d\n",a,b);
```

THE CALLING CODE MUST USE THIS DIFFICULT '&' TO PASS IN THE ADDRESS OF THE UNDERLYING MEMORY LOCATION OF THE VARIABLE

EXAMPLE 21: COMPARE THE OLD (C-STYLE) AND NEW (C++ REFERENCE) STYLE OF SWAPPING TWO VARIABLES

INCREDIBLY COMPLICATED CODE, FOR A REALLY SIMPLE OPERATION!

THE FUNCTION BODY MUST TAKE IN AND DEREFERENCE POINTERS

THE CALLING CODE MUST USE THIS DIFFICULT '&' TO PASS IN THE ADDRESS OF THE UNDERLYING MEMORY LOCATION OF THE VARIABLE

```
// writing the function
double swap(int *a,int *b)
{
   int temp = *a;
   a = b;
   b = temp;
}
// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(&a,&b);
printf("a = %d, b = %d\n",a,b);
```

OH, AND BTW, THIS DOES NOT EVEN INCLUDE ERROR CHECKS FOR NULL VALUES

EXAMPLE 21: COMPARE THE OLD (C-STYLE) AND NEW (C++ REFERENCE) STYLE OF SWAPPING TWO VARIABLES

IN CONTRAST, THE C++ SOLUTION, USING REFERENCES IS FAR SIMPLER

EXAMPLE 21: COMPARE THE OLD (C-STYLE) AND NEW (C++ REFERENCE) STYLE OF SWAPPING TWO VARIABLES

```
// writing the function
double swap(int& a,int& b)
{
  int temp = a;
  a = b;
  b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

INCREDIBLY SIMPLE, EXCEPT FOR ONE TINY LITTLE BIT!

EXAMPLE 21: COMPARE THE OLD (C-STYLE) AND NEW (C++ REFERENCE) STYLE OF SWAPPING TWO VARIABLES

INCREDIBLY SIMPLE, EXCEPT FOR ONE TINY LITTLE BIT!

```
// writing the function
double swap(int& a,int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

A VARIABLE OF TYPE `int&` IS SAID TO BE AN 'INT REFERENCE' OR 'A REFERENCE TO AN INT'

EXAMPLE 21: COMPARE THE OLD (C-STYLE) AND NEW (C++ REFERENCE) STYLE OF SWAPPING TWO VARIABLES

INCREDIBLY SIMPLE, EXCEPT FOR ONE TINY LITTLE BIT!

A VARIABLE OF TYPE `int&` IS SAID TO BE AN 'INT REFERENCE' OR 'A REFERENCE TO AN INT'

```
// writing the function
double swap(int& a,int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

USE AN `int&` EXACTLY AS YOU WOULD AN `int`

EXAMPLE 21: COMPARE THE OLD (C-STYLE) AND NEW (C++ REFERENCE) STYLE OF SWAPPING TWO VARIABLES

INCREDIBLY SIMPLE, EXCEPT FOR ONE TINY LITTLE BIT!

A VARIABLE OF TYPE `int&` IS SAID TO BE AN 'INT REFERENCE' OR 'A REFERENCE TO AN INT'

```
// writing the function
double swap(int& a,int& b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

```
// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

```
a = 5, b = 10
```

USE AN `int&` EXACTLY AS YOU WOULD AN `int`

BUT BE AWARE THAT IT IS PASSED-BY-REFERENCE (LIKE A POINTER)

EXAMPLE 21: COMPARE THE OLD (C-STYLE) AND NEW (C++ REFERENCE) STYLE OF SWAPPING TWO VARIABLES

INCREDIBLY SIMPLE, EXCEPT FOR ONE TINY LITTLE BIT!

A VARIABLE OF TYPE `int&` IS SAID TO BE AN 'INT REFERENCE' OR 'A REFERENCE TO AN INT'

```
// writing the function
double swap(int& a,int& b)
{
   int temp = a;
   a = b;
   b = temp;
}

// calling the function
int a = 5;
int b = 10;
printf("a = %d, b = %d\n",a,b);
swap(a,b);
printf("a = %d, b = %d\n",a,b);
```

```
a = 10, b = 5
```

USE AN `int&` EXACTLY AS YOU WOULD AN `int`

BUT BE AWARE THAT IT IS PASSED-BY-REFERENCE (LIKE A POINTER)

EXAMPLE 22:

RULE #1: A REFERENCE MUST ALWAYS BE INITIALISED

RULE #1: A REFERENCE MUST ALWAYS BE INITIALISED

```
const int& x = 3;
```

X ~~`const int& x;`~~

~~`x = 3;`~~

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example22.cpp
Example22.cpp:10:14: error: declaration of reference variable 'x' requires an initializer
  const int& x;
             ^

1 error generated.
```

EXAMPLE 23:

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

```
int x = 5;
int z = 3;
// Create a reference y, and assign x to it
int& y = x;
cout <<" x = " << x << " y = " << y << " z = " << z << endl;
// Re-assign the reference y so that it is 'equal to' z
y = z;
// Change the value of y
y = 10;
// Will it be x or z that is modified?
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

```
int x = 5;
int z = 3;
```

SET UP 2 SIMPLE INT VARIABLES

```
// Create a reference y, and assign x to it
int& y = x;
cout <<" x = " << x << " y = " << y << " z = " << z << endl;
// Re-assign the reference y so that it is 'equal to' z
y = z;
// Change the value of y
y = 10;
// Will it be x or z that is modified? Answer: x
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

Y IS A REFERENCE TO X

```
int x = 5;
int z = 3;
// Create a reference y, and assign x to it
int& y = x;
cout <<" x = " << x << " y = " << y << " z = " << z << endl;
// Re-assign the reference y so that it is 'equal to' z
y = z;
// Change the value of y
y = 10;
// Will it be x or z that is modified? Answer: x
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

```
int x = 5;
int z = 3;
// Create a reference y, and assign x to it
int& y = x;
cout <<" x = " << x << " y = " << y << " z = " << z << endl;
// Re-assign the reference y so that it is 'equal to' z
y = z;
// Change the value of y
y = 10;
// Did it the x or z that is modified? Answer: x
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

ASSIGN Z TO Y

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

```
int x = 5;
int z = 3;
// Create a reference y, and assign x to it
int& y = x;
cout <<" x = " << x << " y = " << y << " z = " << z << endl;
// Re-assign the reference y so that it is 'equal to' z
y = z;
// Change the value of y
y = 10;
// Will it be x or z that is modified? Answer: x
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

CHANGE THE VALUE OF Y

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

```
int x = 5;
int z = 3;
// Create a reference y, and assign x to it
int& y = x;
cout <<" x = " << x << " y = " << y << " z = " << z << endl;
// Re-assign the reference y so that it is 'equal to' z
y = z;
// Change the value of y
y = 10;

// Will it be x or z that is modified?
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

Answer: x

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

```
int x = 5;
int z = 3;
// Create a reference y, and assign x to it
int& y = x;
cout <<" x = " << x << " y = " << y << " z = " << z << endl;
// Re-assign the reference y so that it is 'equal to' z
y = z;
// Change the value of y
y = 10;

// Will it be x or z that is modified?
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

Answer: x

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

```
// Will it be x or z that is modified?
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

Answer: x

```
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example23.cpp
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
 x = 5 y = 5 z = 3
 x = 10 y = 10 z = 3
```

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

```
int x = 5;
int z = 3;
// Create a reference y, and assign x to it
int& y = x;
cout <<" x = " << x << " y = " << y << " z = " << z << endl;
// Re-assign the reference y so that it is 'equal to' z
y = z;
// Change the value of y
y = 10;
// Will it be x or z that is modified?
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

```
int x = 5;
int z = 3;
// Create a reference y, and assign x to it
int &y = x;
cout <<" x = " << x << " y = " << y << " z = " << z << endl;
// Re assign the reference y so that it is 'equal to' z
y = z;
// Change the value of y
y = 10;
// Will it be x = 10 that is modified? Answer
cout <<" x = " << x << " y = " << y << " z = " << z << endl;
```

WHAT DOES THIS THIS STATEMENT DO?

THIS SIMPLY ASSIGNS THE VALUE OF Z (=3) TO X VIA THE REFERENCE Y

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

```
int x = 5;
int z = 3;
// Create a reference y, and assign it to x
int &y = x;
cout << " x = " << x << " y = " << y << " z = " << z << endl;
// Re assign the reference y so that it is 'equal to' z
y = z;
// Change the value of y
y = 10;
// Will it be x or z that is modified? Answer: x
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

WHAT DOES THIS DO THEN?

THIS SIMPLY ASSIGNS THE VALUE OF Z (=3) TO X

A PRINT STATEMENT HERE WOULD PRINT X = 3 AND Y = 3! REMEMBER 3 IS THE VALUE OF Z

RULE #2: REFERENCE RE-ASSIGNMENTS DON'T DO WHAT YOU THINK THEY WILL

REFERENCES ARE NEVER REASSIGNED, EVEN IF IT SEEMS LIKE THEY HAVE BEEN.

EXAMPLE 24:

RULE #3: MULTIPLE REFERENCES TO THE SAME VALUE CAN EXIST - IF ONE IS MODIFIED, ALL GET MODIFIED

RULE #3: MULTIPLE REFERENCES TO THE SAME VALUE CAN EXIST - IF ONE IS MODIFIED, ALL GET MODIFIED

```
// Create a simple variable
int x = 5;
// Create a reference to that variable: call it reference #1
int& y = x;
// Create another reference to that variable: call it reference #2
int& z = x;

cout <<" x = " << x << " y = " << y << " z = " << z << endl;
// Now modify only reference #2
z = 10;
// Has reference #1 changed? Has the original changed?
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

RULE #3: MULTIPLE REFERENCES TO THE SAME VALUE CAN EXIST - IF ONE IS MODIFIED, ALL GET MODIFIED

```
// Create a simple variable
int x = 5;
// Create a reference to that variable: call it reference #1
int& y = x;
// Create another reference to that variable: call it reference #2
int& z = x;

cout <<" x = " << x << " y = " << y << " z = " << z << endl;
// Now modify only reference #2
z = 10;
// Has reference #1 changed? Has the original changed?
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

RULE #3: MULTIPLE REFERENCES TO THE SAME VALUE CAN EXIST - IF ONE IS MODIFIED, ALL GET MODIFIED

```
// Has reference #1 changed? Has the original changed?
cout <<" x = " << x  << " y = " << y << " z = " << z << endl;
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
  x = 5 y = 5 z = 5
  x = 10 y = 10 z = 10
```

RULE #3: MULTIPLE REFERENCES TO THE SAME VALUE CAN EXIST - IF ONE IS MODIFIED, ALL GET MODIFIED

```
// Has reference #1 changed? Has the original changed?
```

yes and yes

```
cout << " x = " << x << " y = " << y << " z = " << z << endl;
```

```
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
 x = 5 y = 5 z = 5
 x = 10 y = 10 z = 10
```

WE CHANGED Z, WITH THE Z = 10 STATEMENT

RULE #3: MULTIPLE REFERENCES TO THE SAME VALUE CAN EXIST - IF ONE IS MODIFIED, ALL GET MODIFIED

```
// Has reference #1 changed? Has the original changed?
```

yes and yes

```
cout << " x = " << x << " y = " << y << " z = " << z << endl;
```

```
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
x = 5 y = 5 z = 5
x = 10 y = 10 z = 10
```

X, THE ORIGINAL VARIABLE AND Y THE FIRST REFERENCE HAVE ALSO BEEN UPDATED

EXAMPLE 25:

RULE #4: REFERENCES CAN NEVER BE NULL

RULE #4: REFERENCES CAN NEVER BE NULL

WELL - ACTUALLY HERE - THIS ONE IS A TECHNICALITY.

RULE #4: REFERENCES CAN NEVER BE NULL

WELL - ACTUALLY HERE - THIS ONE IS A TECHNICALITY

453. References may only bind to “valid” objects

Section: 8.3.2 [dcl.ref] **Status:** drafting **Submitter:** Gennaro Prota **Date:** 18 Jan 2004

8.3.2 [dcl.ref] paragraph 4 says:

> A reference shall be initialized to refer to a valid object or function. [Note: in particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by dereferencing a null pointer, which causes undefined behavior ...]

THE C++ STANDARD (SECTION 8.3.2) SAYS REFERENCES CAN’T BE NULL

THE C++ STANDARD (SECTION 8.3.2) SAYS REFERENCES CAN'T BE NULL

A REFERENCE SHALL BE INITIALIZED TO REFER TO A VALID OBJECT OR FUNCTION. [NOTE: IN PARTICULAR, A NULL REFERENCE CANNOT EXIST IN A WELL-DEFINED PROGRAM, BECAUSE THE ONLY WAY TO CREATE SUCH A REFERENCE WOULD BE TO BIND IT TO THE "OBJECT" OBTAINED BY DEREFERENCING A NULL POINTER, WHICH CAUSES UNDEFINED BEHAVIOR ...]

RULE #4: REFERENCES CAN NEVER BE NULL

WELL - ACTUALLY HERE - THIS ONE IS A TECHNICALITY.

453. References may only bind to “valid” objects

Section: 8.3.2 [dcl.ref] **Status:** drafting **Submitter:** Gennaro Prota **Date:** 18 Jan 2004

8.3.2 [dcl.ref] paragraph 4 says:

> A reference shall be initialized to refer to a valid object or function. [Note: in particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by dereferencing a null pointer, which causes undefined behavior ...]

THE C++ STANDARD (SECTION 8.3.2) SAYS REFERENCES CAN’T BE NULL

RULE #4: REFERENCES CAN NEVER BE NULL

BUT ACTUALLY YOU CAN END UP WITH A NULL ADDRESS IN A REFERENCE - AND THE COMPILER WILL NOT CATCH THIS.

```
// Create a pointer to an int, initialize to NULL
int * x = NULL;
// Create a reference to that variable: call it reference #1
int& y = *x;
```

RULE #4: REFERENCES CAN NEVER BE NULL

BUT ACTUALLY YOU CAN END UP WITH A NULL ADDRESS IN A REFERENCE - AND THE COMPILER WILL NOT CATCH THIS.

```
// Create a pointer to an int, initialize to NULL
int * x = NULL;
// Create a reference to that variable: call it reference #1
int& y = *x;
// try printing the value - an error will result
// from dereferencing a NULL pointer
cout << y << endl;
```

THIS CODE WILL GIVE A RUNTIME ERROR

RULE #4: REFERENCES CAN NEVER BE NULL

BUT ACTUALLY YOU CAN END UP WITH A NULL ADDRESS IN A REFERENCE - AND THE COMPILER WILL NOT CATCH THIS.

```
// Create a pointer to an int, initialize to NULL
int * x = NULL;
// Create a reference to that variable: call it reference #1
int& y = *x;
// try printing the value - an error will result
// from dereferencing a NULL pointer
cout << y << endl;
```

THIS CODE WILL GIVE A RUNTIME ERROR

```
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example25.cpp
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
 Segmentation fault: 11
```

RULE #4: REFERENCES CAN NEVER BE NULL

MAYBE THIS RULE SHOULD READ:

RULE #4: REFERENCES SHOULD NEVER BE NULL

EXAMPLE 26:

RULE #5: REFERENCES CAN EXIST TO ANY TYPE (INCLUDING POINTERS)

RULE #5: REFERENCES CAN EXIST TO ANY TYPE (INCLUDING POINTERS)

```
// Create a pointer to an int
int *x = new int(5);
// Create a reference to that variable: call it reference #1
int* & y = x;

cout << "contents of the pointer x = " << *x
     << " and contents of the reference pointer y = " << *y << endl;

delete x;
// don't delete y
// because that will lead to the same memory be deallocated twice!
```

RULE #5: REFERENCES CAN EXIST TO ANY TYPE (INCLUDING POINTERS)

```
// Create a pointer to an int
int *x = new int(5);
// Create a reference to that variable: call it reference #1
int* & y = x;

cout << "contents of the pointer x = " << *x
     << " and contents of the reference pointer y = " << *y << endl;

delete x;
// don't delete y
// because that will lead to the same memory be deallocated twice!
```

RULE #5: REFERENCES CAN EXIST TO ANY TYPE (INCLUDING POINTERS)

```
// Create a pointer to an int
int *x = new int(5);
// Create a reference to that variable: call it reference #1
int* & y = x;

cout << "contents of the pointer x = " << *x
     << " and contents of the reference pointer y = " << *y << endl;

delete x;
// don't delete y
// because that will lead to the same memory be deallocated twice!
```

RULE #5: REFERENCES CAN EXIST TO ANY TYPE (INCLUDING POINTERS)

```
// Create a pointer to an int
int *x = new int(5);
// Create a reference to that variable: call it reference #1
int* & y = x;

cout << "contents of the pointer x = " << *x
     << " and contents of the reference pointer y = " << *y << endl;
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example26.cpp
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
contents of the pointer x = 5 and contents of the reference pointer y = 5
```

RULE #5: REFERENCES CAN EXIST TO ANY TYPE (INCLUDING POINTERS)

```
// Create a pointer to an int
int *x = new int(5);
// Create a reference to that variable: call it reference #1
int* & y = x;

cout << "contents of the pointer x = " << *x
     << " and contents of the reference pointer y = " << *y << endl;

delete x;
// don't delete y
// because that will lead to the same memory be deallocated twice!
```

EXAMPLE 27:

RULE #6: NO REFERENCES TO REFERENCES, OR ARRAYS OF REFERENCES

RULE #6: NO REFERENCES TO REFERENCES, OR ARRAYS OF REFERENCES

```
// Create an integer variable
int x = 5;
// Create a reference to that variable: call it reference #1
int & y = x;

// Try to create a reference to the reference - compiler won't allow it
int && z = y;
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example27.cpp
Example27.cpp:15:9: error: 'z' declared as a reference to a reference
  int & & z = y;
        ^
1 error generated.
```

EXAMPLE 28:

BE VERY CAREFUL RETURNING A REFERENCE TO A POINTER FROM A FUNCTION

BE VERY CAREFUL RETURNING A REFERENCE TO A POINTER FROM A FUNCTION

```
int& badFunctionReturnsReferenceToPointer()
{
  // the new is here - but where will the delete be?
  int* x = new int(10);
  return *x;
}


int main()
{

  // Create an integer variable
  int x = badFunctionReturnsReferenceToPointer();
  cout << x << endl;
  // Now no way to delete the pointer - certain memory leak!!

}
```

BE VERY CAREFUL RETURNING A REFERENCE TO A POINTER FROM A FUNCTION

```
int& badFunctionReturnsReferenceToPointer()
{
  // the new is here - but where will the delete be?
  int* x = new int(10);
  return *x;
}


int main()
{

  // Create an integer variable
  int x = badFunctionReturnsReferenceToPointer();
  cout << x << endl;
  // Now no way to delete the pointer - certain memory leak!!

}
```

BE VERY CAREFUL RETURNING A REFERENCE TO A POINTER FROM A FUNCTION

```
int& badFunctionReturnsReferenceToPointer()
{
  // the new is here - but where will the delete be?
  int* x = new int(10);
  return *x;
}


int main()
{

  // Create an integer variable
  int x = badFunctionReturnsReferenceToPointer();
  cout << x << endl;
  // Now no way to delete the pointer - certain memory leak!!

}
```

BE VERY CAREFUL RETURNING A REFERENCE TO A POINTER FROM A FUNCTION

```
int& badFunctionReturnsReferenceToPointer()
{
  // the new is here - but where will the delete be?
  int* x = new int(10);
  return *x;
}


int main()
{

  // Create an integer variable
  int x = badFunctionReturnsReferenceToPointer();
  cout << x << endl;
  // Now no way to delete the pointer - certain memory leak!!

}
```

EXAMPLE 29:

NEVER RETURN A REFERENCE TO A STACK VARIABLE FROM A FUNCTION

NEVER RETURN A REFERENCE TO A STACK VARIABLE FROM A FUNCTION

```
int& badFunctionReturnsReferenceToStackVariable()
{
  // the variable x is on the stack of this function
  int x(10);
  return x;
  // NO! x will cease to exist when the function returns!
}

int main()
{

  // Create an integer variable
  int x = badFunctionReturnsReferenceToStackVariable();
  cout << x << endl;
  // x will be invalid memory by this point!
}
```

NEVER RETURN A REFERENCE TO A STACK VARIABLE FROM A FUNCTION

```
int& badFunctionReturnsReferenceToStackVariable()
{
  // the variable x is on the stack of this function
  int x(10);
  return x;
  // NO! x will cease to exist when the function returns!
}

int main()
{

  // Create an integer variable
  int x = badFunctionReturnsReferenceToStackVariable();
  cout << x << endl;
  // x will be invalid memory by this point!
}
```

NEVER RETURN A REFERENCE TO A STACK VARIABLE FROM A FUNCTION

```
int& badFunctionReturnsReferenceToStackVariable()
{
  // the variable x is on the stack of this function
  int x(10);
  return x;
  // NO! x will cease to exist when the function returns!
}

int main()
{

  // Create an integer variable
  int x = badFunctionReturnsReferenceToStackVariable();
  cout << x << endl;
  // x will be invalid memory by this point!
}
```

NEVER RETURN A REFERENCE TO A STACK VARIABLE FROM A FUNCTION

```
int& badFunctionReturnsReferenceToStackVariable()
{
  // the variable x is on the stack of this function
  int x(10);
  return x;
  // NO! x will cease to exist when the function returns!
}

int main()
{

  // Create an integer variable
  int x = badFunctionReturnsReferenceToStackVariable();
  cout << x << endl;
  // x will be invalid memory by this point!
}
```

NEVER RETURN A REFERENCE TO A STACK VARIABLE FROM A FUNCTION

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example29.cpp
Example29.cpp:10:10: warning: reference to stack memory associated with local variable 'x'
      returned [-Wreturn-stack-address]
  return x;
         ^
1 warning generated.
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
10
```

COMPILING WITH THE `-Wall` OPTION WILL ACTUALLY WARN ABOUT THIS!

NEVER RETURN A REFERENCE TO A STACK VARIABLE FROM A FUNCTION

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ g++ -Wall Example29.cpp
Example29.cpp:10:10: warning: reference to stack memory associated with local variable 'x'
      returned [-Wreturn-stack-address]
  return x;
         ^
1 warning generated.
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
10
```

IN THIS CASE THE OUTPUT ACTUALLY WORKS OK, BUT THAT'S PURELY BY LUCK!