Outline 2017/03/10

- Communications et al.
- Basic grammar:
  - Expressions, types, declarations, statements
  - Casting
  - Functions
  - Heap and stack memory, the pointers
  - Arrays (operator [ ])
- References and more on pointers
- More on functions
- const specifier
- C++ and strings
- References for today:

  learncpp.com : ch. 2 (all), 3.1→3.6, 5.2, 5.5, 5.7, 5.8, 6.7 – 6.12, 7.1 – 7.4, 7.9, 17.2

Communications et al.

1) course planning and following lessons
2) chmod +x
3) last exercise

Communications et al.

1) course planning and following lessons

March 2017

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
			1 Ash Wednesday	2	3 OK	4
5	6	7	8	9	10 OK	11 Purim
12 Daylight Savings Begins	13	14	15	16	17 St. Patrick's Day OK	18
19	20 Spring Begins	21	22	23	24 OK (?)	25
26	27	28	29	30	31 NO	

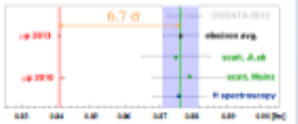
FAMU – fundamental physics

FAMU: high precision measurement of the muonic hydrogen hyperfine splitting

E. Mocchiutti (Emiliano.Mocchiutti@ts.infn.it) on behalf of the FAMU Collaboration: A. Adamczak, G. Baccolo, D. Bakalov, G. Baldazzi, R. Bertoni, M. Bonesini, V. Bonvicini, G. Campana, R. Carbone, T. Cervi, F. Chignoli, M. Clemenza, L. Colace, A. Curioni, M. Danailov, P. Danev, I. D'Antone, A. De Bari, C. De Vecchi, M. De Vincenzi, M. Furini, F. Fuschino, K.S. Gadedjisso-Tossou, D. Giuffarti, A. Iaciofano, K. Ishida, D. Iugovaz, C. Labanti, V. Maggi, A. Margotti, M. Marsiaki, R. Mazza, S. Meneghini, A. Menegolli, E. Mocchiutti, M. Moretti, G. Morgante, R. Nardò, M. Nastasi, J. Niemela, E. Previtali, R. Ramponi, A. Rachevski, L.P. Rignanese, M. Rossella, P.L. Rossi, F. Somma, M. Stoilov, L. Stoychev, A. Tomasell, L. Tortora, A. Vacchi, E. Vallazza, G. Zampa and M. Zuffa.

1. Experimental errors, new physics or misgauged Rydberg constant?

The proton radius is extremely hard to measure with high precision and with excellent control of systematics.

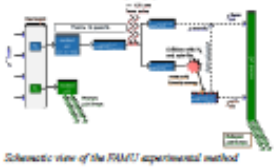


Proton radius measurements: discrepancy between electron measurements and muon spectroscopy at PSI [1].

A shrinking of the proton in presence of a muon could signify the existence of a **previously unknown fundamental force** - one that acts between protons and muons but not between protons and electrons. Another possibility is a **misgauged Rydberg constant**, a factor that goes in the QED calculations of differences between atomic energy levels.

2. FAMU experimental method

FAMU will realize a spectroscopic measurement of the hyperfine splitting (hfs) in the 1S state of muonic hydrogen $\Delta E^{hfs}(\mu p)_{1s}$ with **unprecedented precision** – $\delta\Delta/\Delta < 10^{-3}$ – providing crucial information on proton structure and muon- nucleon interaction [2]. The method is the following: muonic hydrogen atoms (μp) form in an appropriate mixture of hydrogen and a higher-Z gas. A μp in the ground state, after absorbing a photon of energy equal to the hfs resonance-energy, is very quickly de-excited in subsequent collision with other molecules. At the exit of the collision the μp is accelerated by ~2/3 of the excitation energy, taken away as kinetic energy. The observable is the time distribution of the characteristic X-rays emitted from the muonic atoms formed by μ transfer (which is kinetic energy dependent [3]) from hydrogen to the atom of the admixture gas $(\mu p) + Z \rightarrow (\mu Z)^* + p$ and its response to variations of the laser radiation wavelength [4].

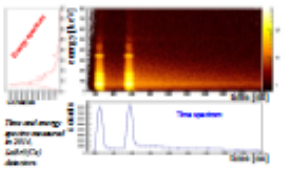


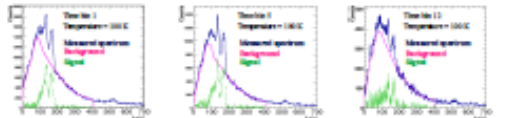
Schematic view of the FAMU experimental method

An intense muon beam, like the RIKEN-RAL one (UK), is needed to achieve the needed statistics and precision. The planning is to: 1) optimize the gas filling of the target and it operating conditions (2014 run) 2) measure the transfer rate from μp to Oxygen (2016 run) 3) measure $\Delta E^{hfs}(\mu p)_{1s}$ and derive the proton Zemach radius, run with the laser to drive the hfs transition (expected in 2018).

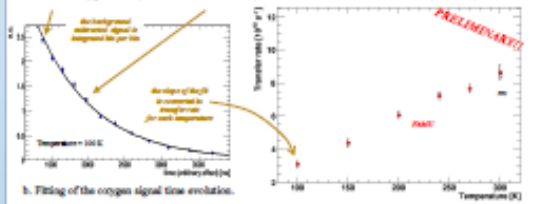
3. Detectors characterization and transfer rate to Oxygen

The experimental apparatus includes a precise **fiber-SiPMT beam hodoscope**, a crown of **eight LaBr3 crystals** and **four HPGe** detectors for detection of emitted characteristic X-rays. FAMU target is kept at 40 bars, permitting about $3{\cdot}10^4$ μ/s stop in the target forming muonic hydrogen, as compared to the $4{\cdot}10^3$ μ/s flux from the continuous muon source of PSI, a gain of nearly two orders of magnitude. During first validation tests at the beam delivery Port 4 of the RIKEN-RAL facility, the detection system and the beam condition allowed a perfectly satisfactory background situation [2] and an excellent time and energy X-rays reconstruction. Measurements taken in 2016 confirm the **energy dependent muon transfer rate to Oxygen.**





a. Oxygen lines (~133 keV and ~150 keV) reconstruction in different time bins (2016 run).



b. Fitting of the oxygen signal time evolution.

c. For the **first time** ever the transfer rate is measured as function of the temperature.

A cryogenic temperature controlled target was used to study the time evolution of Oxygen X-ray lines. At each of the six chosen temperatures the delayed Oxygen signal is measured after a background estimation and subtraction. This procedure is performed in several time bins delayed respect the muon arrival (panel a). The time evolution is then fitted with an exponential function (panel b) and the transfer rate is finally evaluated (panel c).

4. Outlook and future measurements

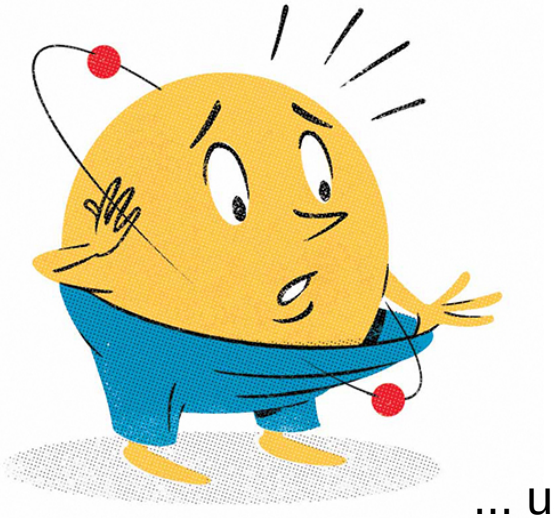
While finalizing the preliminary measurements, the laser measurements is prepared by:

- finishing the realization of the **tunable mid-infrared laser source** with the needed characteristics - wavelength 6785 nm (44.22 THz), line width ~0.07 nm (450 MHz), tunability ±10 nm (120 GHz), operating at repetition rate 50Hz with 20-30 ns long pulse, pulse energy >2 mJ. The laser is based on non-linear optics using direct difference frequency generation in non-oxide nonlinear crystals and using a Q-switched single longitudinal mode Nd:YAG laser (1064 nm) and a tunable narrowband Cr:forsterite laser operating at 1260 nm, pumped by another YAG laser [5].
- realizing an **optical multi-pass reflecting cavity** meeting strict geometrical and optical constraints to maximize the spin-flip transition probability. Our specifically-designed mirrors will have reflectivity better than 99.95%.
- modifying the 2016 cryogenic target to include optical path and cavity.

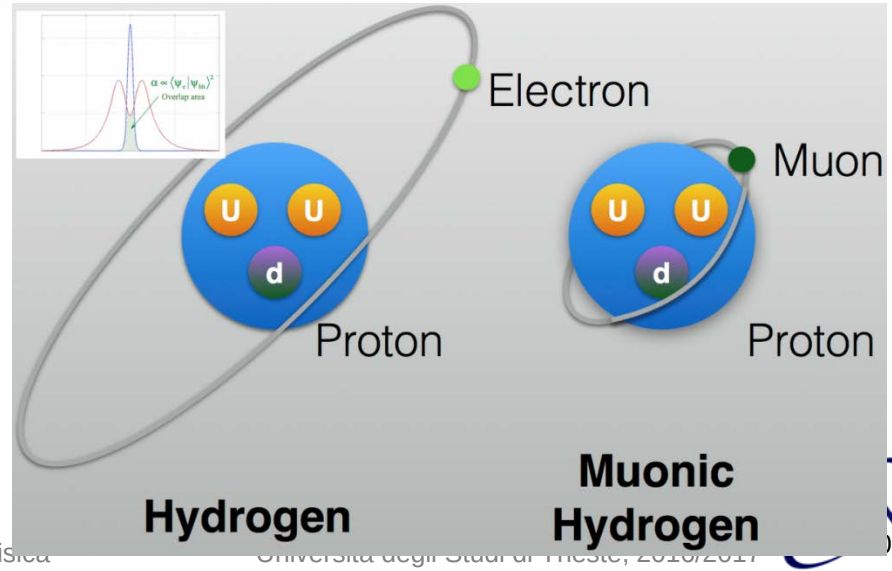
A set of preliminary measurements with the laser will be carried out in 2017, aiming at 2018 for the measurement of the Zemach radius.

Bibliography

Study of proton size and fundamental constants...



... using muonic atoms



Communications et al.

1) course planning and following lessons

April 2017

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
						1 April Fool's Day
2	3	4	5	6	7 OK	8
9 Palm Sunday	10 Passover	11	12	13	14 OK (?)	15
16 Easter Sunday	17	18	19	20	21 OK	22 Earth Day
23 30	24	25	26	27	28 OK	29

Communications et al.

1) course planning and following lessons

May 2017

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	1 May Day	2	3	4	5 OK Cinco de Mayo	6
7	8	9	10	11	12 OK	13
14 Mother's Day	15	16	17	18	19 OK	20 Armed Forces Day
21	22 Victoria Day	23	24	25	26 OK	27
28	29 Memorial Day	30	31		June 2nd holiday in Italy	

Communications et al.

1) course planning and following lessons
2) chmod +x

- scripts can be run with:
  - `|prompt>source script.sh`
  - `|prompt>. script.sh`
  - `|prompt>./script.sh` (but first you must make script.sh executable) it opens a new session

script.sh is NOT an executable and can not be compiled, hence it cannot be run with «./test.sh»...

Communications et al.

1) course planning and following lessons
2) chmod +x

- scripts can be run with:
  - `|prompt>source script.sh`
  - `|prompt>. script.sh`
  - `|prompt>./script.sh` (but first you must make script.sh executable) it opens a new session

script.sh is NOT an executable and can not be compiled, hence it cannot be run with «./test.sh»... UNLESS we tell the OS it can be run as an executable by giving (once forever) the command

bash> chmod +x test.sh

Communications et al.

1) course planning and following lessons
2) chmod +x
3) last time exercises

Examples and exercises

```
|Emi@marte tmp>mkdir /tmp/prova
|Emi@marte tmp>cd /tmp/prova/
/tmp/prova
|Emi@marte prova>mkdir dir1
|Emi@marte prova>mkdir dir2
|Emi@marte prova>mkdir dir1/bin
|Emi@marte prova>mkdir dir2/bin
|Emi@marte prova>cd dir1/bin/
/tmp/prova/dir1/bin
|Emi@marte bin>nano test.sh

#!/bin/bash
echo "this is test in dir 1";

|Emi@marte bin>chmod a+x test.sh
|Emi@marte bin>cd ..
|Emi@marte dir1>cd ..
|Emi@marte prova>cd dir2/bin/
/tmp/prova/dir2/bin
```

Examples and exercises

```
|Emi@marte bin>nano test.sh

#!/bin/bash
echo "this is test in dir 2

|Emi@marte bin>chmod a+x test.sh
|Emi@marte bin>cd ..
|Emi@marte dir2>cd ..
|Emi@marte prova>cd dir1/bin/
/tmp/prova/dir1/bin
|Emi@marte bin>chmod a+x test.sh
|Emi@marte bin>cd ..
|Emi@marte dir1>cd ..
|Emi@marte prova>export PATH=/tmp/prova/dir1/bin:$PATH
|Emi@marte prova>test.sh
this is test in dir 1
|Emi@marte prova>export PATH=/tmp/prova/dir2/bin:$PATH
|Emi@marte prova>test.sh
this is test in dir 2
|Emi@marte prova>
```

Examples and exercises

|Emi@marte prova>test.sh > pippo1.txt

|Emi@marte prova>test.sh > pippo2.txt

|Emi@marte prova>test.sh > pippo3.txt

|Emi@marte prova>ls pippo*

pippo3.txt pippo2.txt pippo1.txt

|Emi@marte prova> **`for file in `ls pippo*`; do echo $file; mv $file `basename $file .txt`_test.txt ; done`**

pippo3.txt

pippo2.txt

pippo1.txt

|Emi@marte prova>ls pippo*

pippo3_test.txt pippo2_test.txt pippo1_test.txt

where:

- for syntax is: **for** variable_name **in** *list_of_something*; **do** *do_something* $variable_name; **done**

- ` means execute inline the command that follows

- **basename:** basename pippo.txt .txt gives “pippo”

Expressions and Operators

http://www.neurophys.wisc.edu/comp/docs/notes/not017.html

Arithmetic operations/functions

.	Fortran	Example	C/C++	Example	Comment
Add	+	A=B+C	+	a=b+c;	.
Subtract	-	A=B-C	-	a=b-c;	.
Multiply	*	A=B*C	*	a=b*c;	.
Divide	/	A=B/C	/	a=b/c;	.
Modulus	MOD	A=MOD(B,C)	%	a=b%c;	ints only in C, reals possible in Fortran
Power	**	A=B**C	pow()	a=pow(b,c);	C/C++ function not intrinsic, provided via math.h

	Fortran	C++	
• post-increment:	a = a+1	a++	use a and then increment a
• pre-increment:		++a	increment a and use the result

```
cout << " post increment " << a++ << "\n";    if a is 0 it prints 0
cout << " pre increment  " << ++a << "\n";    if a is 0 it prints 1
```

Expressions and Operators

Relational and Logical Operators

.	Fortran	Example	C/C++	Example	Comment
Equal to	.EQ.	IF(A.EQ.B)...	==	if(a==b)...	.
Not Equal to	.NE.	IF(A.NE.B)...	!=	if(a!=b)...	.
Less Than	.LT.	IF(A.LT.B)...	<	if(a<b)...	.
Greater Than	.GT.	IF(A.GT.B)...	>	if(a>b)...	.
Less Than or Equal to	.LE.	IF(A.LE.B)...	<=	if(a<=b)...	.
Greater Than or Equal to	.GE.	IF(A.GE.B)...	>=	if(a>=b)...	.
Logical Not	.NOT.	IF(.NOT.A)...	!	if(!a)...	.
Logical AND	.AND.	IF(A.AND.B)...	&&	if(a&&b)...	.
Logical OR	.OR.	IF(A.OR.B)...	||	if(a||b)...	.

Expressions and Operators

```
int a = 10;
int b = 5;
if ( a == b ) cout << “ a is equal to b \n”;
if ( a != b ) cout << “ a and b differ \n”;
if ( a < b ) cout << “ a is smaller than b \n”;
if ( !a ) cout << “a is different from zero \n”;

if ( a < b && a > b ) cout << “something is wrong \n”;

if ( a == b || a > b ) cout << “same as a >= b \n”;
```

Expressions and Operators

Bitwise Operators

.	Fortran	Example	C/C++	Example	Comment
Bitwise AND	IAND	IAND(N,M)	&	n&m	.
Bitwise OR	IOR	IOR(N,M)	|	n|m	.
Bitwise Exclusive OR	IEOR	IEOR(N,M)	^	n^m	.
Bitwise 1's Complement	INOT	INOT(N)	~	~n	.
Bitwise Left Shift	ISHFT	ISHFT(N,M) (M > 0)	<<	n<<m	n shifted left by m bits
Bitwise Right Shift	ISHFT	ISHFT(N,M) (M < 0)	>>	n>>m	n shifted right by m bits

Expressions and Operators

Mathematical Functions

Note: In C/C++, must include the header file "math.h" to use these functions. Angles must be specified in radians for these functions. "n/a" means "not available".

.	Fortran	Example	C/C++	Example	Comment
Sine	SIN	SIN(R)	n/a	n/a	Single Precision
Sine	DSIN	DSIN(R)	sin	sin(r)	Double Precision
Cosine	COS	COS(R)	n/a	n/a	Single Precision
Cosine	DCOS	DCOS(R)	cos	cos(r)	Double Precision
Tangent	TAN	TAN(R)	n/a	n/a	Single Precision
Tangent	DTAN	DTAN(R)	tan	tan(r)	Double Precision
Arc Sine	ASIN	ASIN(R)	n/a	n/a	Single Precision
Arc Sine	DASIN	DASIN(R)	asin	asin(r)	Double Precision
Arc Cosine	ACOS	ACOS(R)	n/a	n/a	Single Precision
Arc Cosine	DACOS	DACOS(R)	acos	acos(r)	Double Precision
Arc Tangent	ATAN	ATAN(R)	n/a	n/a	Single Precision
Arc Tangent	DATAN	DATAN(R)	atan	atan(r)	Double Precision
Hyperbolic Sine	SINH	SINH(R)	n/a	n/a	Single Precision
Hyperbolic Sine	DSINH	DSINH(R)	sinh	sinh(r)	Double Precision
Hyperbolic Cosine	COSH	COSH(R)	n/a	n/a	Single Precision
Hyperbolic Cosine	DCOSH	DCOSH(R)	cosh	cosh(r)	Double Precision
Hyperbolic Tangent	TANH	TANH(R)	n/a	n/a	Single Precision
Hyperbolic Tangent	DTANH	DTANH(R)	tanh	tanh(r)	Double Precision

Expressions and Operators

Mathematical Functions

Note: In C/C++, must include the header file "math.h" to use these functions. Angles must be specified in radians for these functions. "n/a" means "not available".

.	Fortran	Example	C/C++	Example	Comment
Sine	SIN	SIN(R)	n/a	n/a	Single Precision
Sine	DSIN	DSIN(R)	sin	sin(r)	Double Precision
Cosine	COS	COS(R)	n/a	n/a	Single Precision
Cosine	DCOS	DCOS(R)	cos	cos(r)	Double Precision
Tangent	TAN	TAN(R)	n/a	n/a	Single Precision
Tangent	DTAN	DTAN(R)	tan	tan(r)	Double Precision
Arc Sine	ASIN	ASIN(R)	n/a	n/a	Single Precision
Arc Sine	DASIN	DASIN(R)	asin	asin(r)	Double Precision
Arc Cosine	ACOS	ACOS(R)	n/a	n/a	Single Precision
Arc Cosine	DACOS	DACOS(R)	acos	acos(r)	Double Precision
Arc Tangent	ATAN	ATAN(R)	n/a	n/a	Single Precision
Arc Tangent	DATAN	DATAN(R)	atan	atan(r)	Double Precision
Hyperbolic Sine	SINH	SINH(R)	n/a	n/a	Single Precision
Hyperbolic Sine	DSINH	DSINH(R)	sinh	sinh(r)	Double Precision
Hyperbolic Cosine	COSH	COSH(R)	n/a	n/a	Single Precision
Hyperbolic Cosine	DCOSH	DCOSH(R)	cosh	cosh(r)	Double Precision
Hyperbolic Tangent	TANH	TANH(R)	n/a	n/a	Single Precision
Hyperbolic Tangent	DTANH	DTANH(R)	tanh	tanh(r)	Double Precision

```
#include <math.h>
...

float angle = 0.3;
float sine = sin(angle);
float cosine = cos(angle);
```

Statements

Blocks: { }

variables LIVE in the block (and sub-blocks) and DIE outside:

```
int myGlobal = 100;
int main(){
   int a = 1;
   // here a is 1 and myGlobal is 100
   {
       int b = 3;
       // here a is still 1 and myGlobal is 100
       // here b is 3
   }
   // here a is 1, myGlobal is 100
   // but b is UNDEFINED!
   return 0;
}
// here myGlobal is 100, a and b are UNDEFINED
```

Conditional Statements

if

usage: if (*condition*) *statement* [[else if (*condition*) *statement*] else *statement*]

```
int i = 0;
if ( i < 100 ) cout << i << "\n";
if ( i < 100 ){
   cout << i << "\n";
}
int n = 3;
int m = 0;
if ( n > 4 ){
   cout << n << "\n";
} else if ( n <= 4 && n > 0 ){
   cout << m++ << "\n";
} else {
   cout << " n <= 0 \n";
}
```

Iteration Statements

while

usage: while (condition) statement

```
int i = 0;
while ( i < 100 ) i++;

bool isFull = false;
while ( (i >= 99 && i < 200) || isFull ){
   i++;
   // other statements which could change isFull boolean
   // to “true”
}
```

Iteration Statements

for

usage: for (for-init-statement;condition;expression) statement

```
for (int i = 0; i < 100 ; i++) cout << i << "\n";
for (int i = 0; i < 100 ; i++){
   cout << i << "\n";
}

int m = 0;
for (int i = 100; i > 0; i--){
   cout << i << "\n";
   cout << m++ << "\n";
}
```

Iteration Statements

break , continue (to be used carefully)

break: stop and exit the loop

continue: skip all the remaining statements of the block and pass to the following iteration

```
for (int i = 0; i < 100 ; i++){
   if ( i == 50 ) continue;
   if ( i > 75 ) break;
   cout << i << "\n";
}
```

this will print all numbers from 0 to 75 (included) “50” excluded:

1 2 3 ... 49 51 ... 75

Types

bool: 1 bytes
char: 1 bytes
wchar_t: 4 bytes
short: 2 bytes
int: 4 bytes
long: 4 bytes
long long: 8 bytes
float: 4 bytes
double: 8 bytes
long double: 12 bytes

FORTRAN	C/C++
byte	unsigned char
integer*2	short int
integer	long int or int
integer iabc(2,3)	int iabc[3][2];
logical	long int or int
logical*1	bool (C++, One byte)
real	float
real*8	double
real*16	long double
complex	struct{float realnum; float imagnum;}
double complex	struct{double dr; double di;}
character*6 abc	char abc[6];
character*6 abc(4)	char abc[4][6];
parameter	#define *PARAMETER value*

The size of a given data type is dependent on the compiler and/or the computer architecture. On most 32-bit machines, a char is 1 byte, a bool is 1 byte, a short is 2 bytes, an int is 4 bytes, a long is 4 bytes, a float is 4 bytes, and a double is 8 bytes

Types

typesSize.cpp

```
#include <iostream>
using namespace std;
int main(){
    cout << "bool:\t\t" << sizeof(bool) << " bytes\n";
    cout << "char:\t\t" << sizeof(char) << " bytes\n";
    cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes\n";
    cout << "short:\t\t" << sizeof(short) << " bytes\n";
    cout << "int:\t\t" << sizeof(int) << " bytes\n";
    cout << "long:\t\t" << sizeof(long) << " bytes\n";
    cout << "long long:\t\t" << sizeof(long long) << " bytes\n";
    cout << "float:\t\t" << sizeof(float) << " bytes\n";
    cout << "double:\t\t" << sizeof(double) << " bytes\n";
    cout << "long double:\t" << sizeof(long double) << " bytes\n";
    return 0;
}
```

“void” type

“void” is a value of a function type that means nothing is returned.

It is the type of functions known as procedures, functions that do NOT return a value.

Example:

```
void Print(){
 cout << ...
}
```

Casting

Definition: a variable of a certain type can be forced to be of a different type – warning: this can result in information loss!

C++: static_cast<TypeA> varTypeB

```
int nDogs = 10;
int nChips = 5;
float chipsPerDog = static_cast<float>(nChips) /
                                    static_cast<float>(nDogs);

float chipsPerDog = nChips / nDogs;    WRONG! it gives 0.

float chipsPerDog = static_cast<float>(nChips /
                                    nDogs);    WRONG! it gives 0.
```

Casting

Warning: CASTING does NOT mean ROUNDING!

```
#include <math.h>
float eFlux = 1.23456789;
float pFlux = 9.87654321;
cout << " integer eFlux " << static_cast<int>(eFlux);
```

integer eFlux 1

```
cout << " integer pFlux " << static_cast<int>(pFlux);
```

integer pFlux 9

```
cout << " integer eFlux " << round(eFlux);
```

integer eFlux 1

```
cout << " integer pFlux " << round(pFlux);
```

integer pFlux 10

Casting, usual (C) way to do

Definition: a variable of a certain type can be forced to be of a different type – warning: this can result in information loss!

C: (TypeA)varTypeB

```
int nDogs = 10;
int nChips = 5;
float chipsPerDog = (float)(nChips) /
                                (float)(nDogs);

float chipsPerDog = nChips / nDogs;    WRONG! it gives 0.

float chipsPerDog = (float)(nChips /
                                nDogs);    WRONG! it gives 0.
```

Declaration

A declaration is the definition of a certain type variable, function, etc. (`int a, float c, ...`)

A declaration is formed by:

1. specifier (optional argument), determine a property of the object. Most common specifiers: `virtual, extern, const`
2. *type*, determine the type of the object (`float, int, void`,...)
3. **declarator** and name
   - `* name` (operator pointer)
   - `* const name` (operator constant pointer)
3. name and **declarator**
   - `name[]` (operator array)
   - `name()` (operator funcion)

Declarations

variable declaration is mandatory!!

A declaration is the definition of a certain type variable, function, etc. (`int a, float c, ...`)

A declaration is formed by:

1. specifier (optional argument), determine a property of the object. Most common specifiers: `virtual, extern, const`
2. *type*, determine the type of the object (`float, int, void`,...)
3. **declarator** and name
   - `* name` (operator pointer)
   - `* const name` (operator constant pointer)
3. name and **declarator**
   - `name[]` (operator array)
   - `name()` (operator funcion)

Declaration

```
specifier type declaration name
```

Declaration examples:

```
int a;
const float mean = 10.4582;
int m[3] = {0,4,8};
int *n;
extern int countStudents(int,int,float);
virtual void Print();
```

Declaration

```
specifier type declaration name
```

Declaration to be understood must be read from the right to the left, from the name backward.

Declaration examples:

```
int a; // a is a integer
float *t = new float(9.); // t is a pointer to a
  float variable
char *name; // name is a pointer to a char
  variable
int countStudents(int, int, float); //
  countStudents is a function returning an
  integer
```

The “const” specifier

The const specifier states that the value of an object can NOT be modified, it is constant.

Examples (look at constDeclarations.cpp) :

```
int const nStudents = 10; // nStudents is a constant integer (also written: const int nStudents)
nStudents = 5; // WRONG! (does not compile)


int const *i = new int(8); // i is a pointer to a constant integer: i can be only 8 but we can move the variable in the memory, pointer is not constant!
int *const i = new int(8); // i is a constant pointer to an integer variable: i can be any value but we cannot move the variable in memory, pointer is constant
int const *const i = new int(8); // i is a constant pointer to a constant integer: neither the pointer nor the value can be changed
```

Functions

Functions are pieces of code performing a specific task.

When declaring a function we need:

- *returning type of the function*
- function name
- **function arguments** (if any)

Example:

```
int countStudents(int nStudents, int nMissing, float
  missingProbability){
  int nStudentsHere;
  // [...] block performing the task
  return nStudentsHere;
}
```

Functions

Functions can be declared in a program and defined in another one, example:

File main.cpp:

```
int countStudents(int, int, float);
int main(){
  // block doing something...
   int nOfStudents = countStudents(nTotal,nNotHere,probability);
   return ;
}
```

File count.cpp:

```
int countStudents(int nStudents, int nMissing, float
   missingProbability){
   int nStudentsHere;
  // [...] block performing the task
   return nStudentsHere;
}
```

Functions

To compile:

```
|prompt>g++ -c count.cpp
|prompt>g++ -c main.cpp
|prompt>g++ -o studentCounter count.o main.o
```

compilation

linking

Function arguments can have pre-defined values:

```
int countStudents(int nStudents = 10, int
  nMissing = 0, float missingProbability = 0.){
  int nStudentsHere;
  // [...] block performing the task
  return nStudentsHere;
}
```

Function in this case can be called this way:

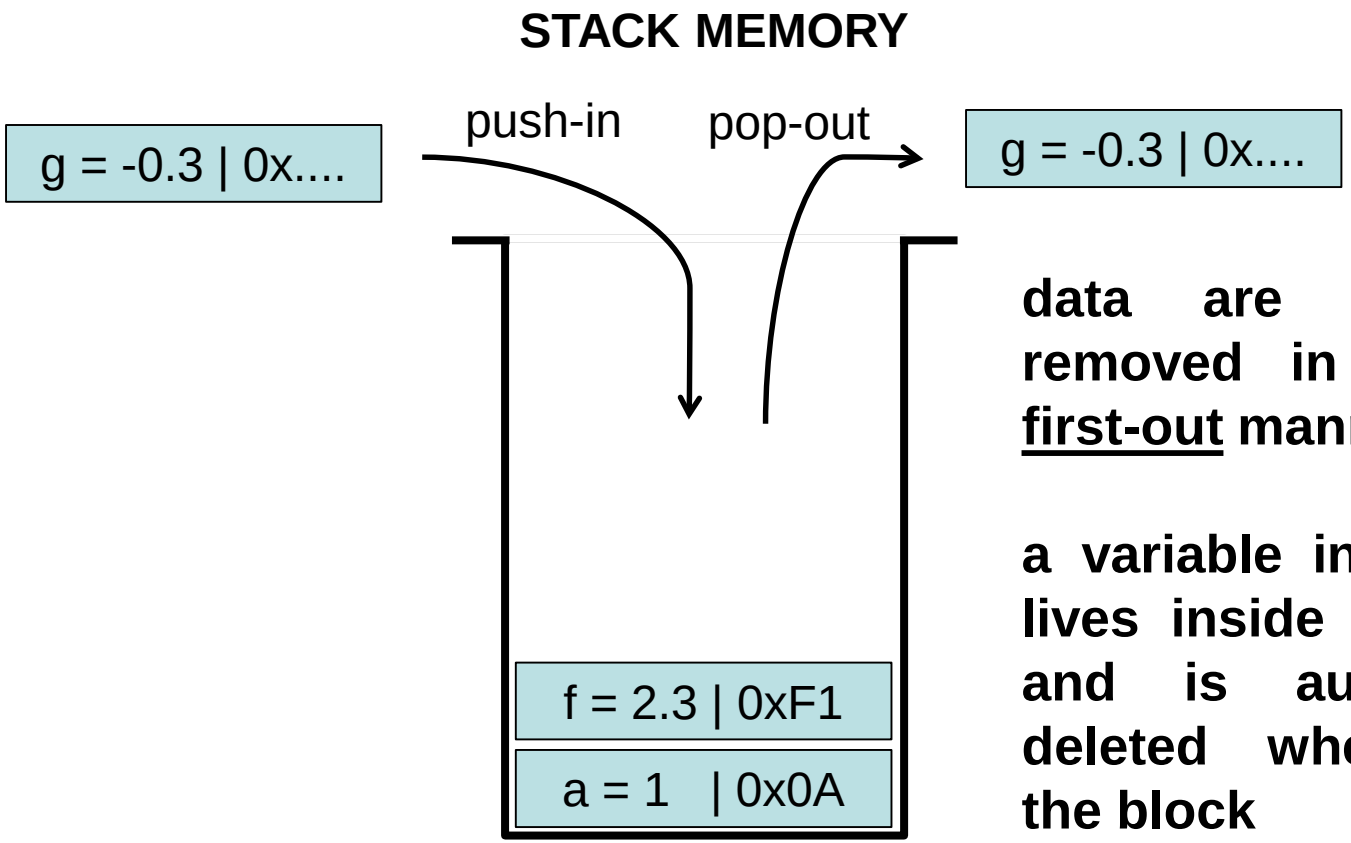
```
int nOfStudents = countStudents();
```

Stack and Heap memory

Disclaimer: VERY SIMPLIFIED AND NOT REAL (*IT*) DESCRIPTION!

When a program is executed a certain RAM is assigned to it by the OS.

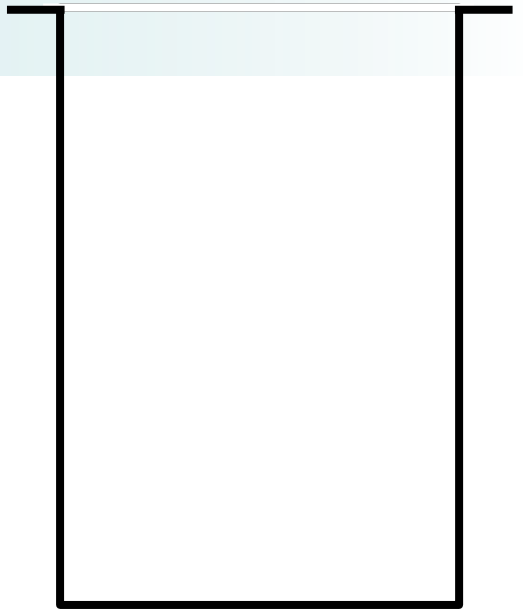
This RAM is splitted into STACK memory and HEAP memory.



data are added or removed in a last-in-first-out manner

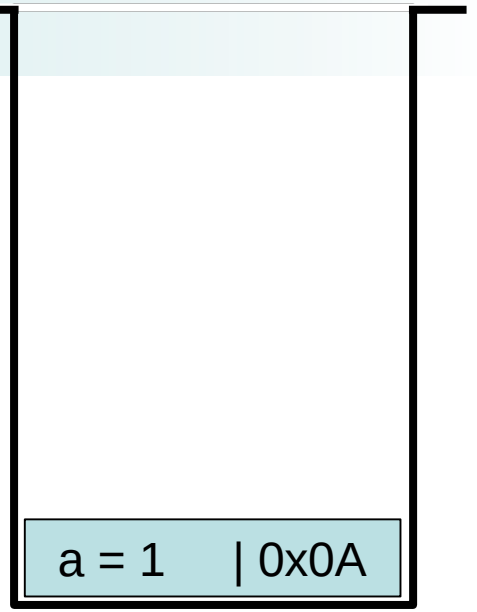
a variable in the stack lives inside the blocks and is automatically deleted when exiting the block

Stack memory



```
int main(){
    int a = 1;
    if ( a > 0 ){
        float f = 2.3;
        int m = 4;
        float c = static_cast<float>(m) / f;
        for ( int j = 0 ; j < 2 ; j++){
              c += 0.12;
        }
        cout << " here c is " << c << "\n";
    }
    return 0;
}
```

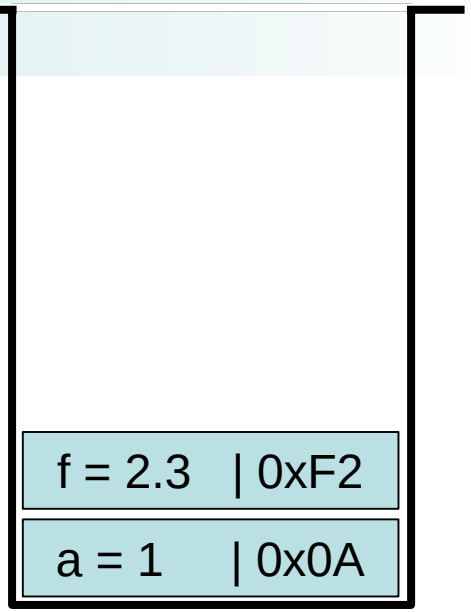
Stack memory




```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

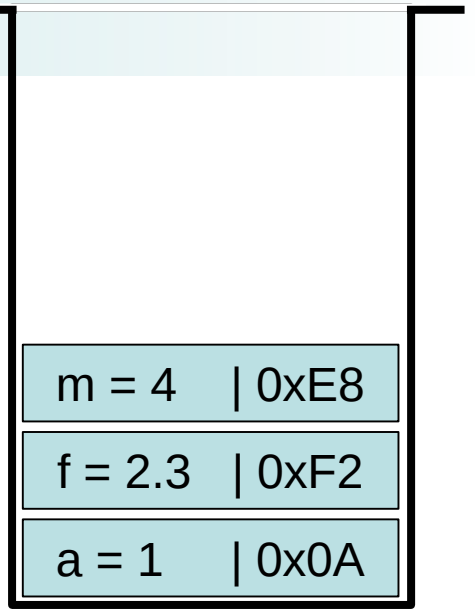
Stack memory

```
int main(){
   int a = 1;
   if ( a > 0 ){
      float f = 2.3;
      int m = 4;
      float c = static_cast<float>(m) / f;
      for ( int j = 0 ; j < 2 ; j++){
            c += 0.12;
      }
      cout << " here c is " << c << "\n";
   }
   return 0;
}
```

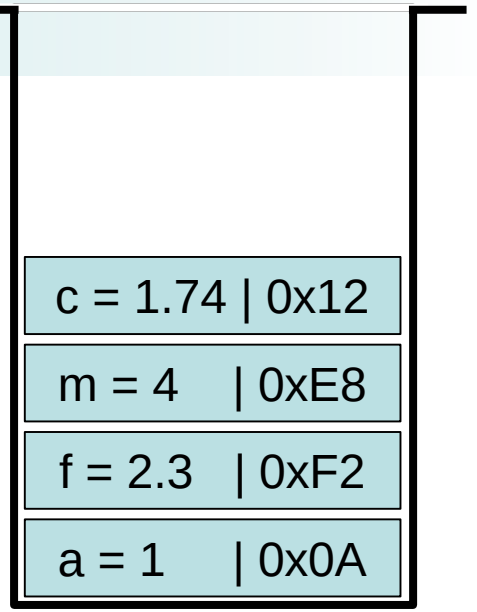


Stack memory

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

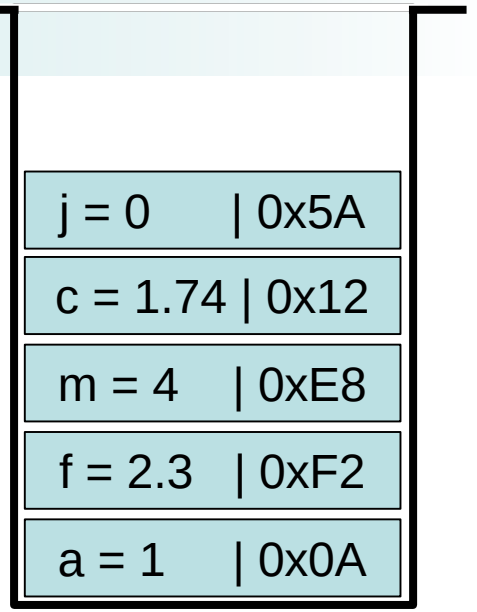


Stack memory



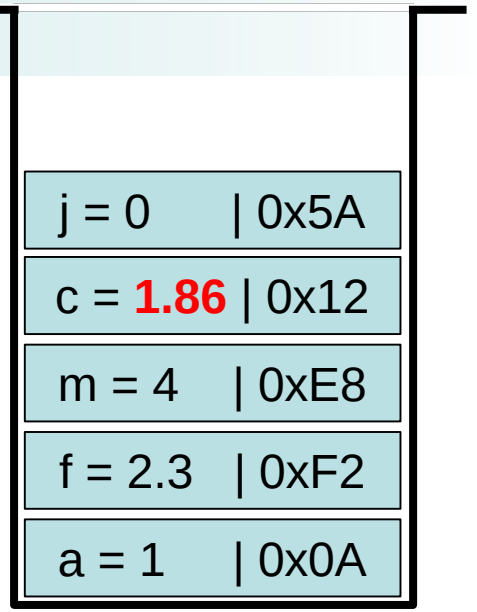

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Stack memory



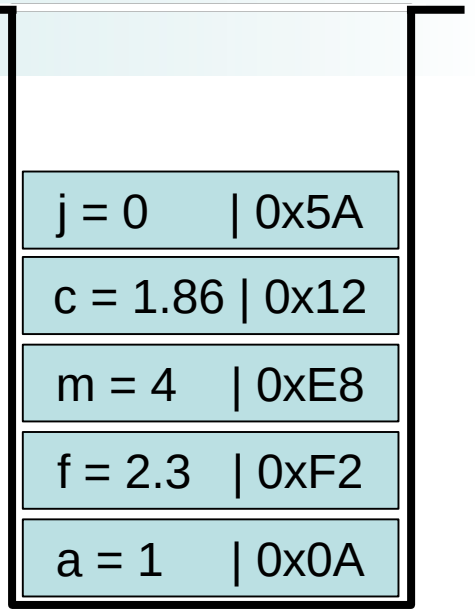

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Stack memory



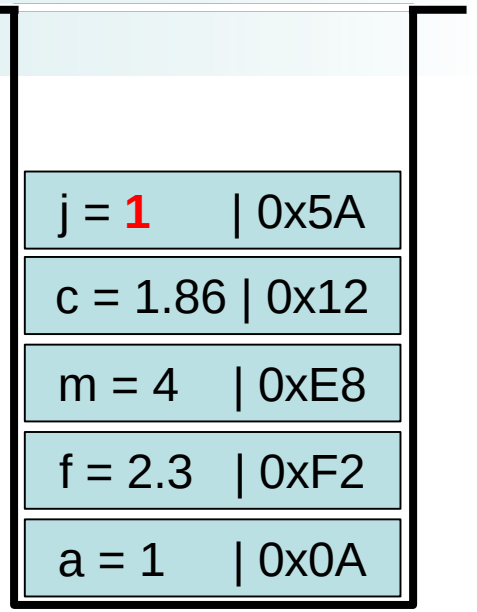

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Stack memory



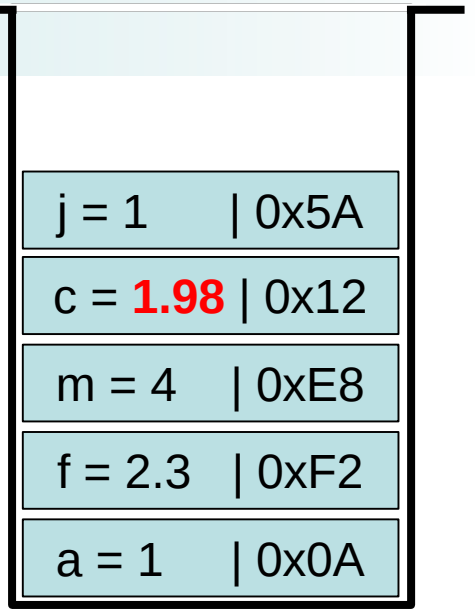

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Stack memory



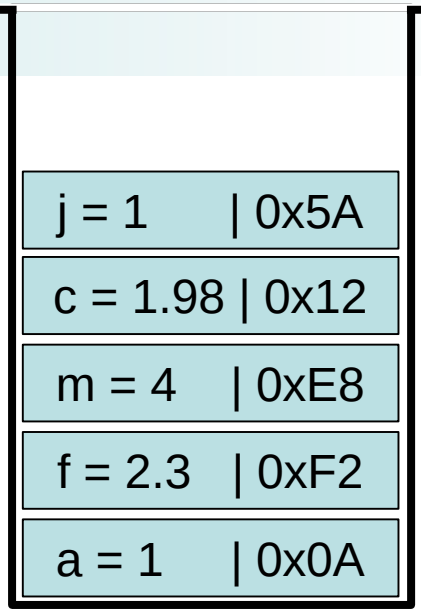

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Stack memory



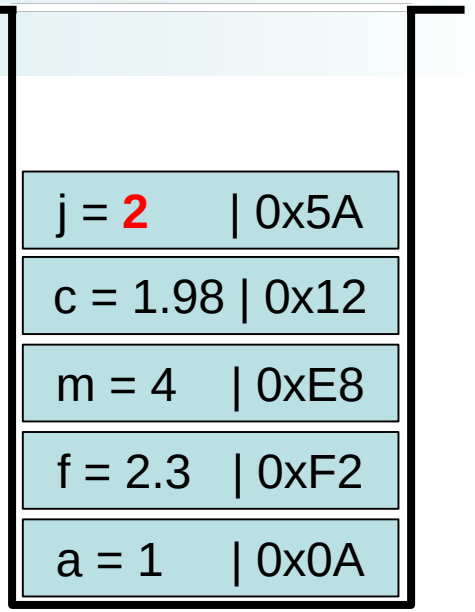

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Stack memory



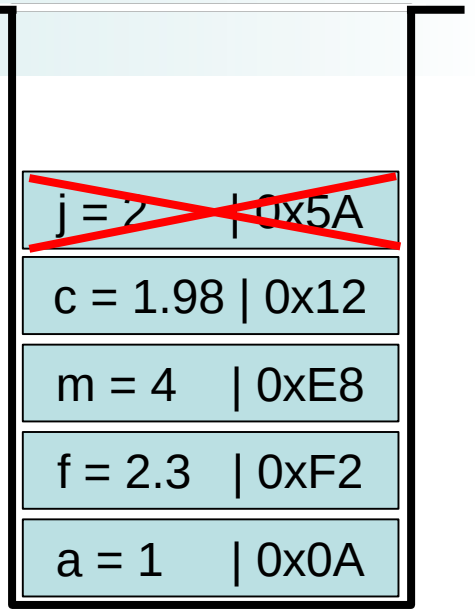

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Stack memory



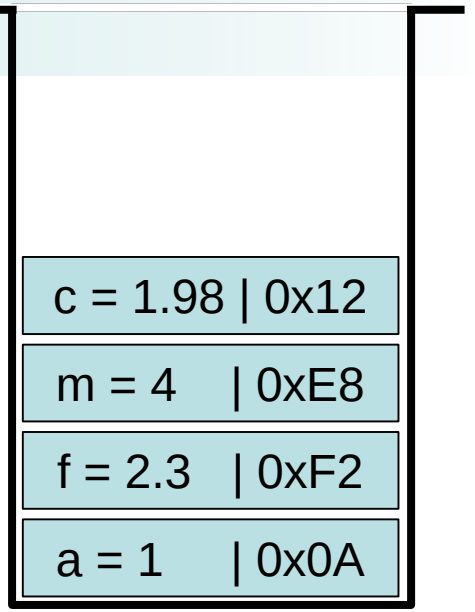

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Stack memory




```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

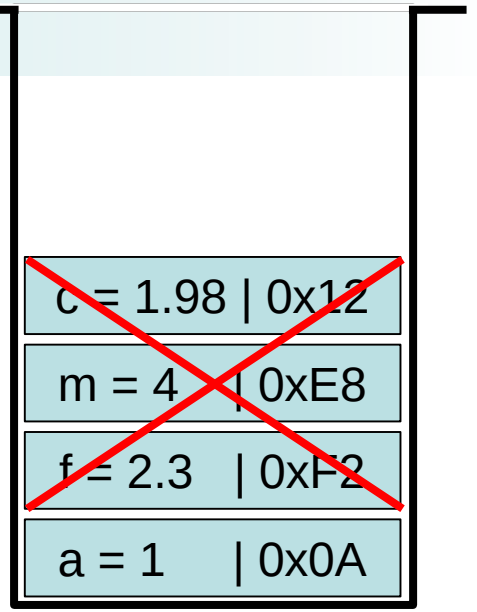
Stack memory



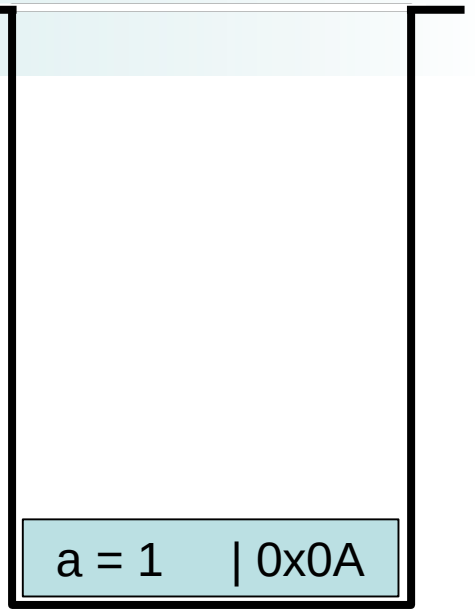

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Stack memory

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

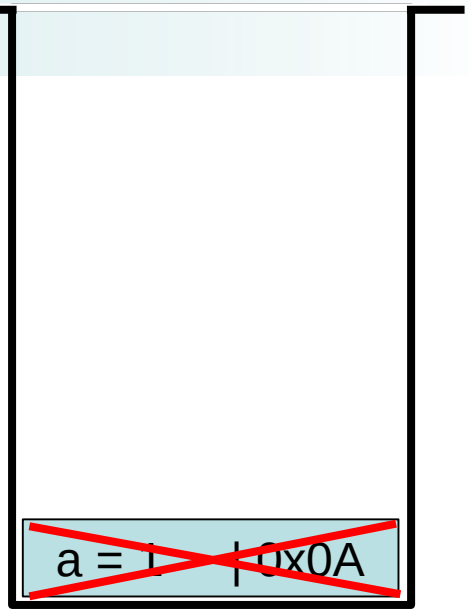


Stack memory



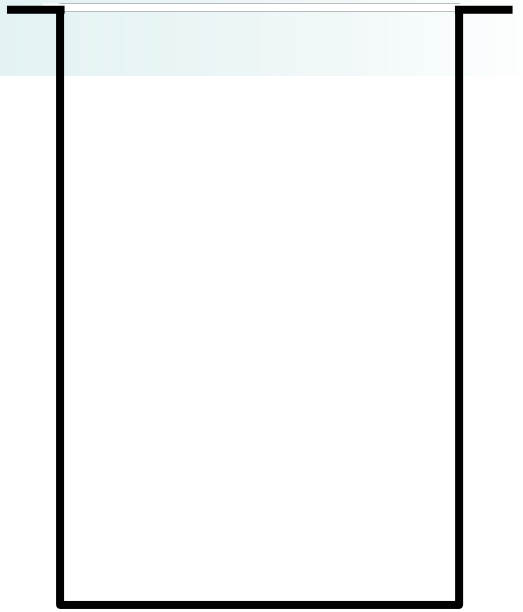

```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Stack memory




```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Stack memory

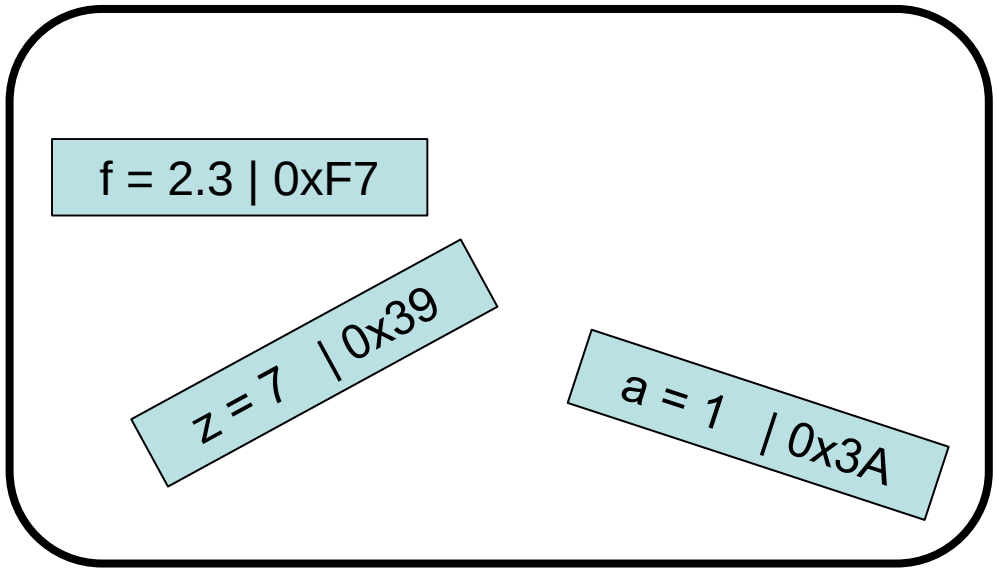


```
int main(){
   int a = 1;
   if ( a > 0 ){
       float f = 2.3;
       int m = 4;
       float c = static_cast<float>(m) / f;
       for ( int j = 0 ; j < 2 ; j++){
             c += 0.12;
       }
       cout << " here c is " << c << "\n";
   }
   return 0;
}
```

Heap memory

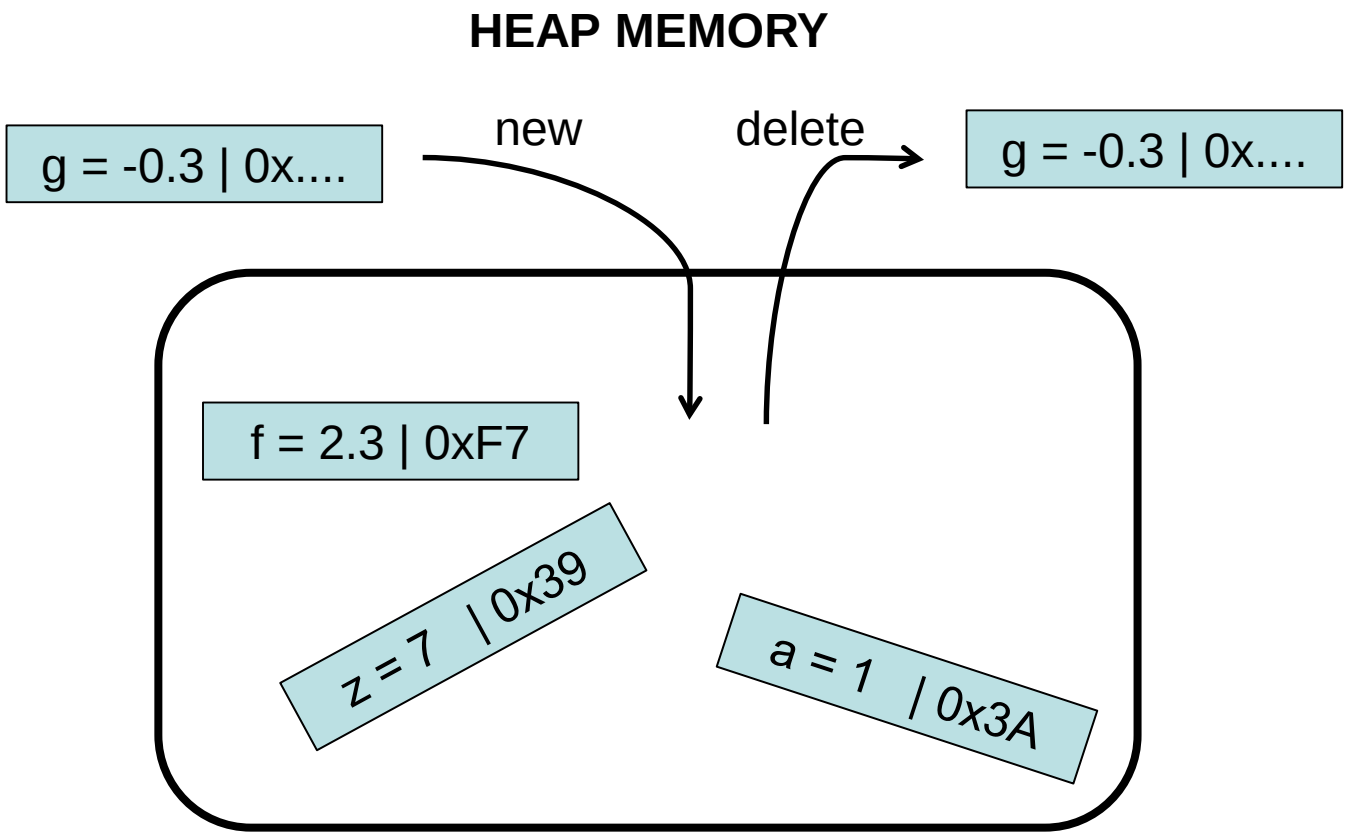
Big memory requests can be satisfied by allocating portions from a large pool of memory, the heap. At any given time, some parts of the heap are in use, while some are "free" (unused) and thus available for future allocations.

HEAP MEMORY



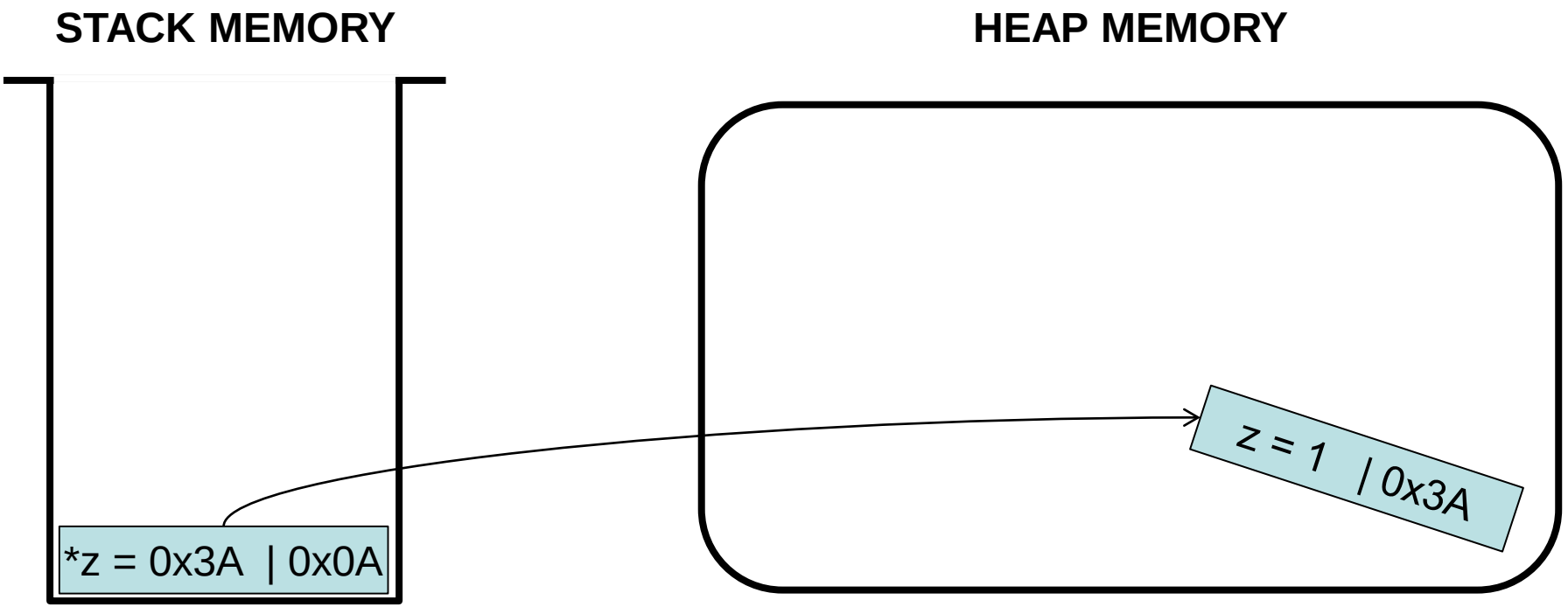
Heap memory – memory allocation

- to allocate heap memory we use the operator “**new**”
- to free heap memory we use the operator “**delete**”



Heap memory – need pointers

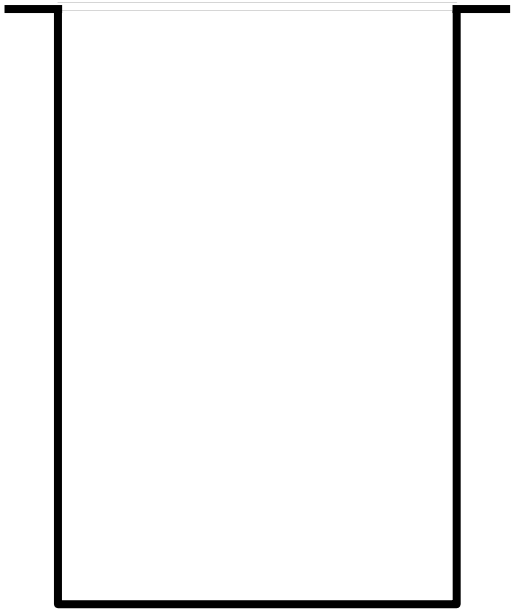
to access data in the heap we need a POINTER in the stack

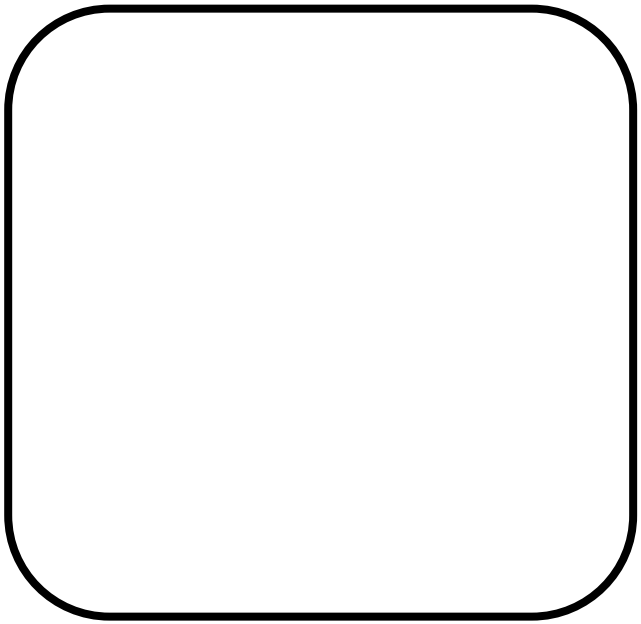



```
int *z = new int(1);
```

WARNING: heap memory is not freed automatically!

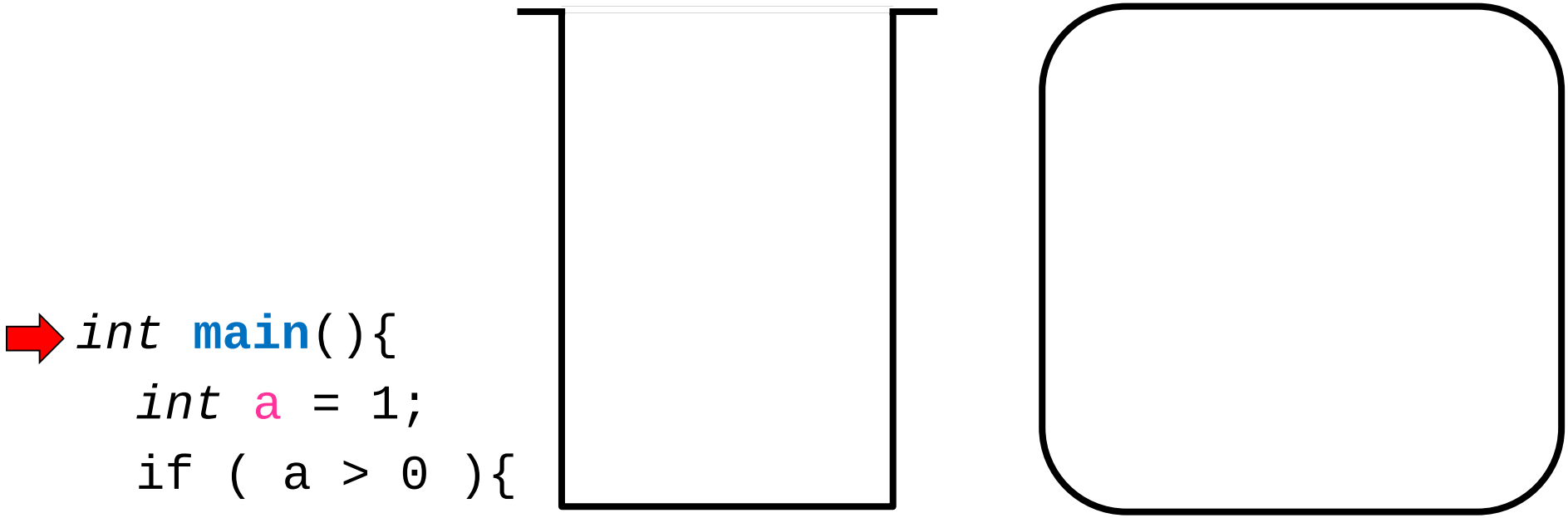
Heap and stack memory





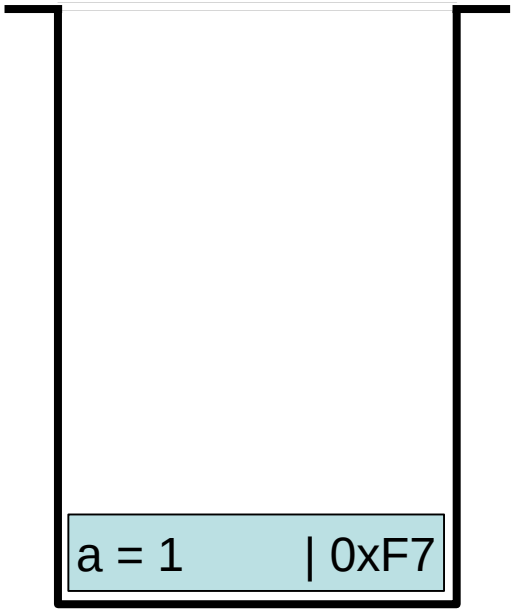
```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
       delete z;
       z = 0;
   }
   return 0;
}
```

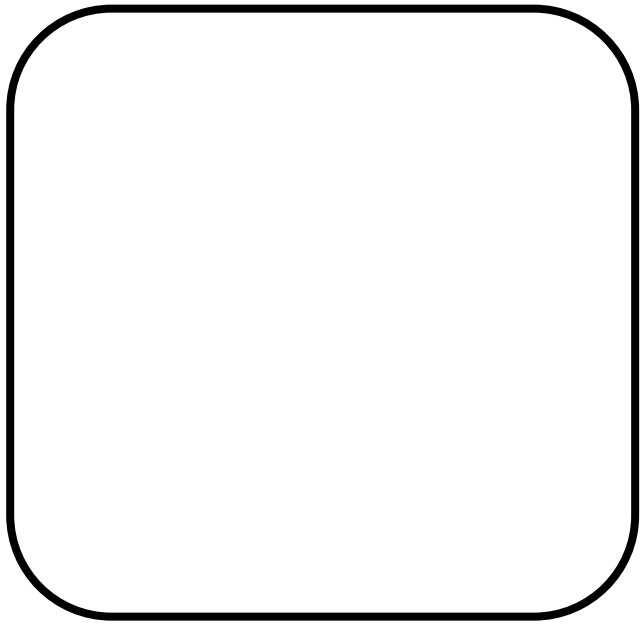
Heap and stack memory



```
int main(){
    int a = 1;
    if ( a > 0 ){
        int *z = new int(1);
        // do something
        delete z;
        z = 0;
    }
    return 0;
}
```

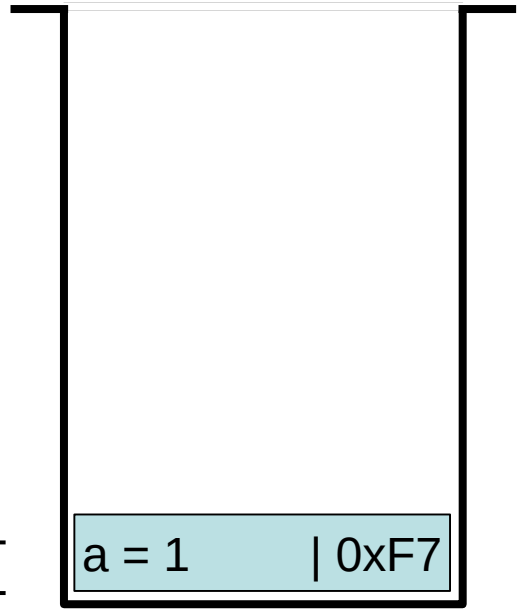
Heap and stack memory

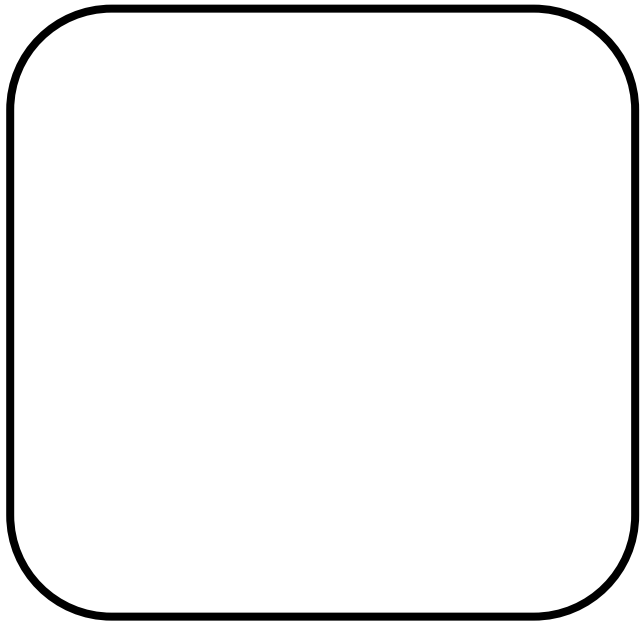




```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
       delete z;
       z = 0;
   }
   return 0;
}
```

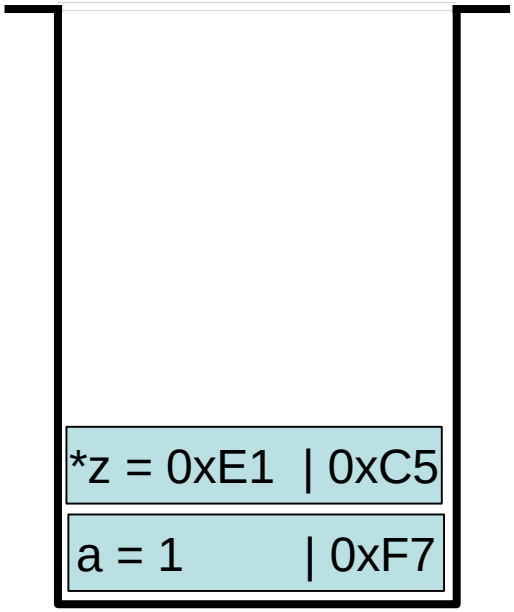
Heap and stack memory

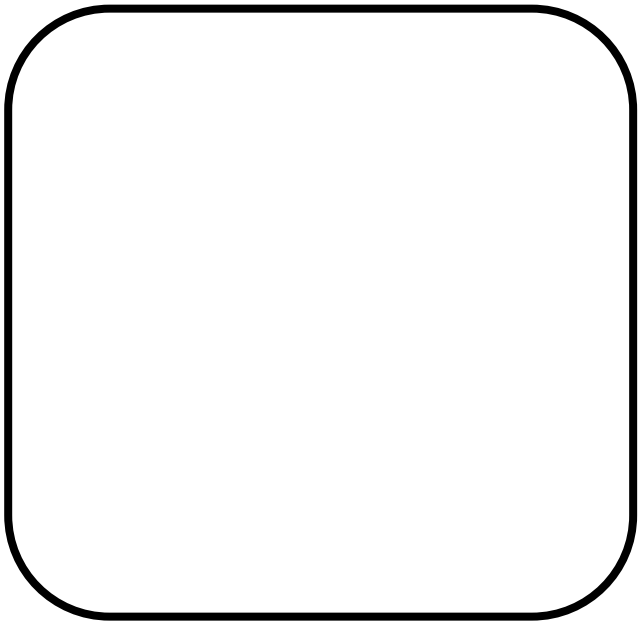




```
int main(){
   int a = 1;
   if ( a > 0 ){
      int *z = new int(1);
      // do something
      delete z;
      z = 0;
   }
   return 0;
}
```

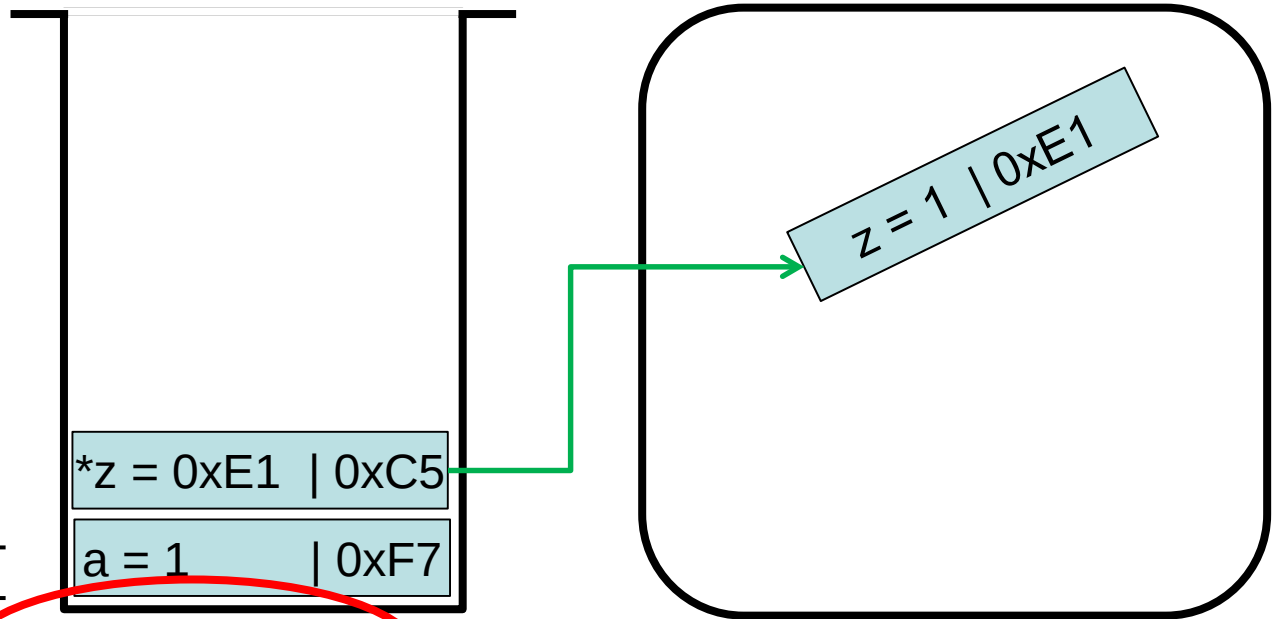
Heap and stack memory





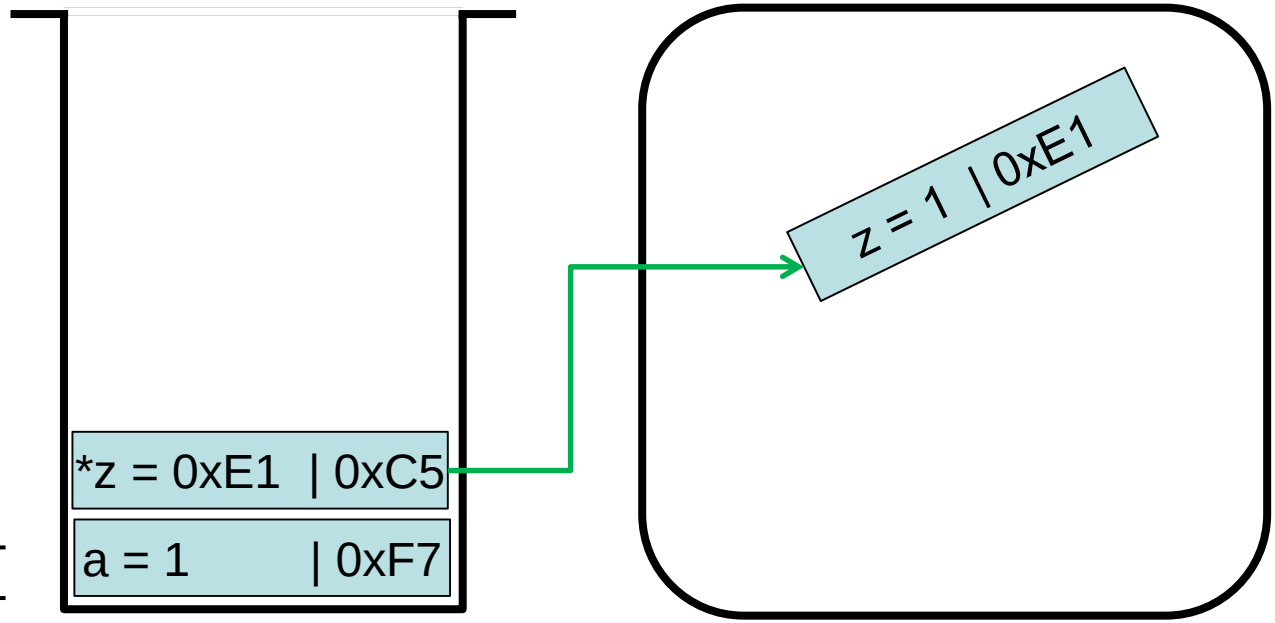
```
int main(){
   int a = 1;
   if ( a > 0 ){
      int *z = new int(1);
      // do something
      delete z;
      z = 0;
   }
   return 0;
}
```

Heap and stack memory



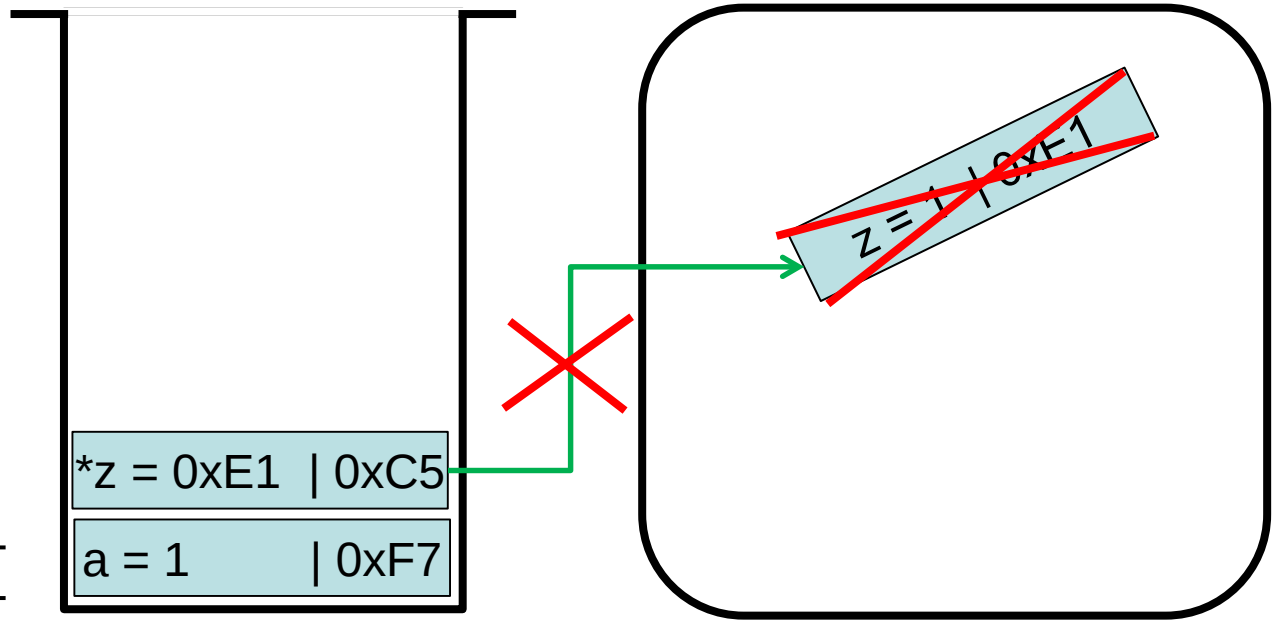

```
int main(){
   int a = 1;
   if ( a > 0 ){
      int *z = new int(1);
      // do something
      delete z;
      z = 0;
   }
   return 0;
}
```

Heap and stack memory



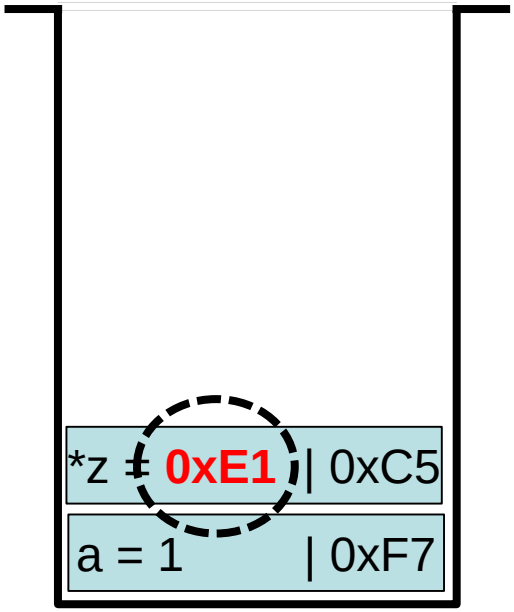

```
int main(){
   int a = 1;
   if ( a > 0 ){
      int *z = new int(1);
      // do something
      delete z;
      z = 0;
   }
   return 0;
}
```

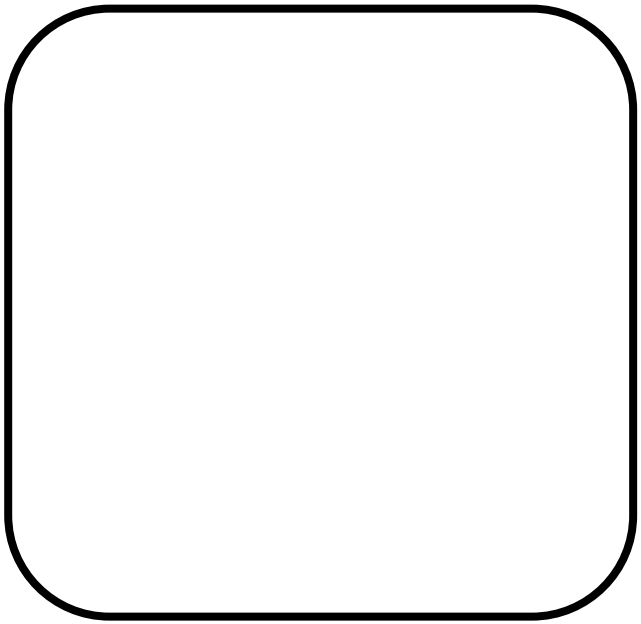
Heap and stack memory




```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
       delete z;
       z = 0;
   }
   return 0;
}
```

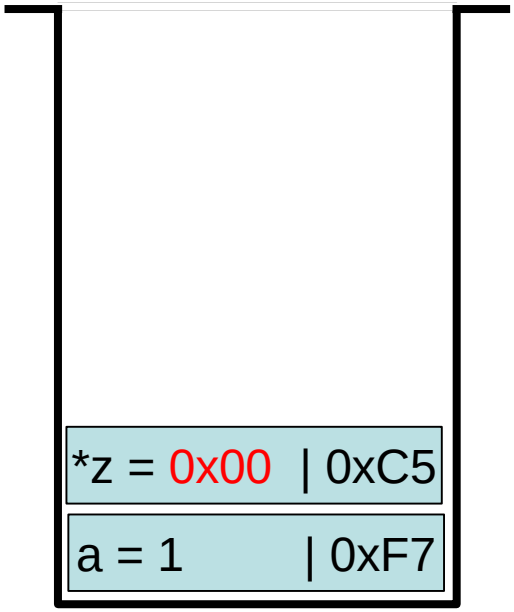
Heap and stack memory

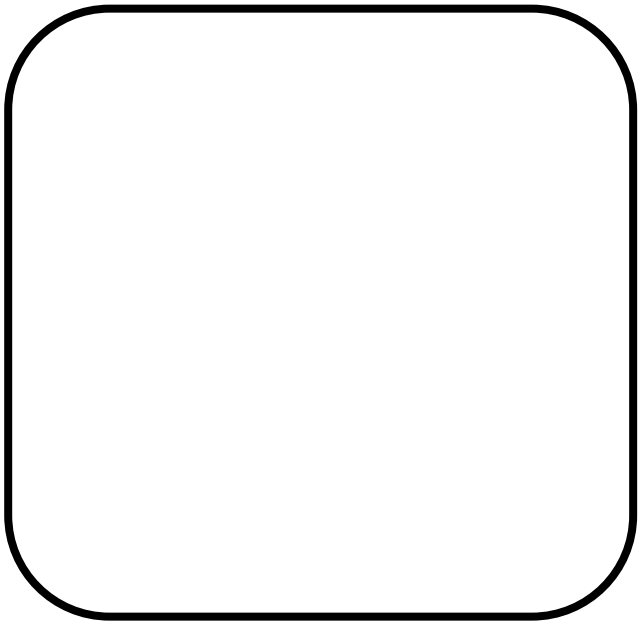




```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
       delete z;
       z = 0;
   }
   return 0;
}
```

Heap and stack memory

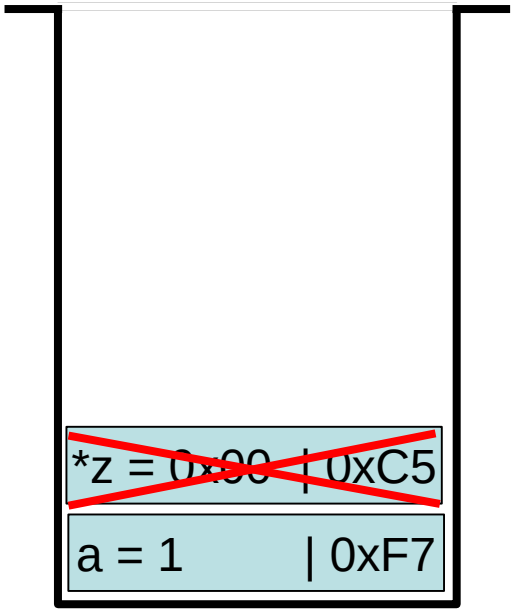


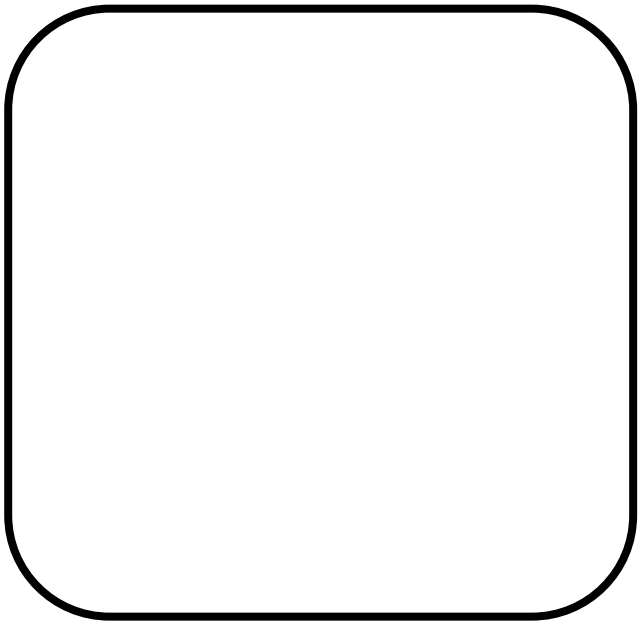


correggi

```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
       delete z;
       z = 0;
   }
   return 0;
}
```

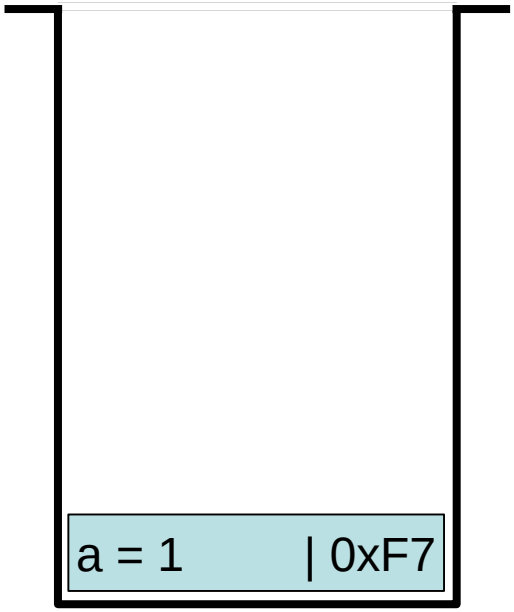
Heap and stack memory

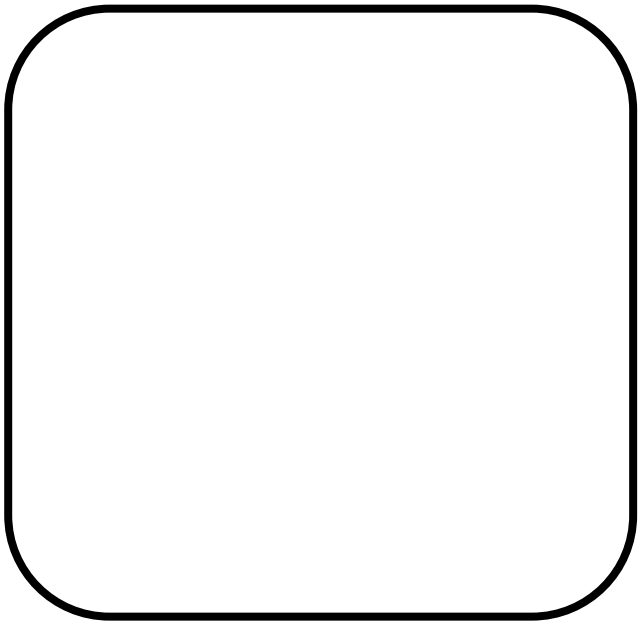




```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
       delete z;
       z = 0;
   }
   return 0;
}
```

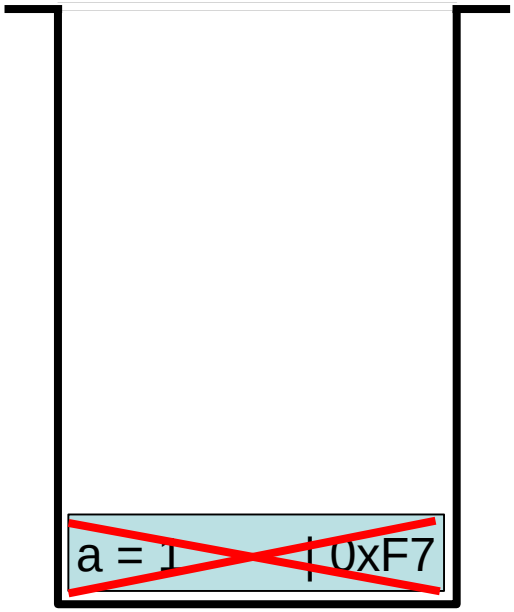
Heap and stack memory

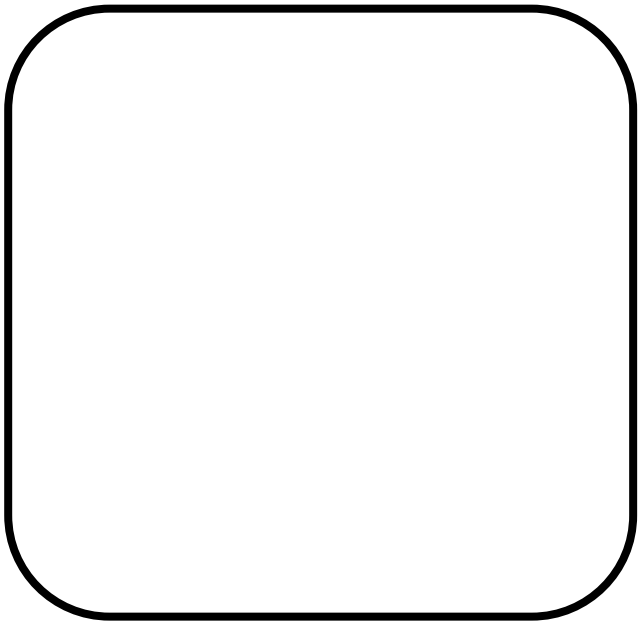




```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
       delete z;
       z = 0;
   }
   return 0;
}
```

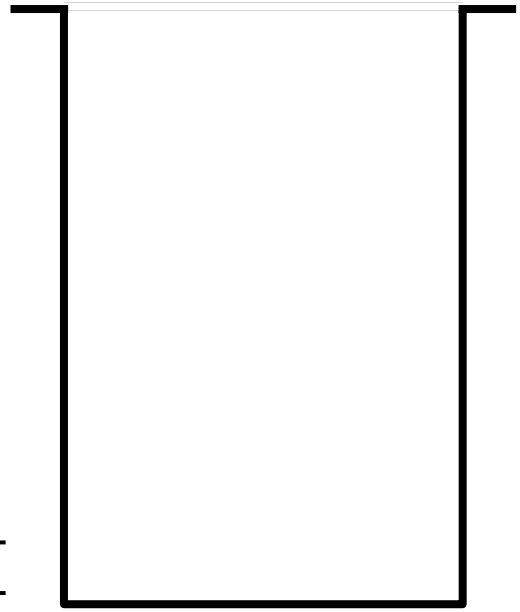
Heap and stack memory

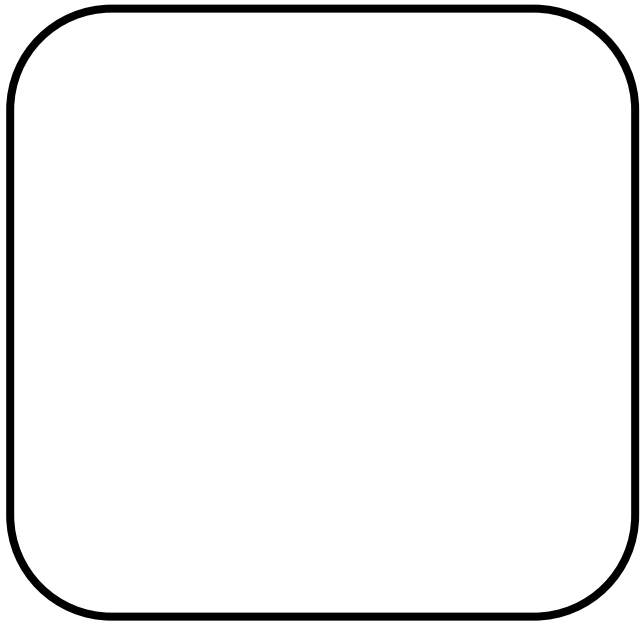




```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
       delete z;
       z = 0;
   }
   return 0;
}
```

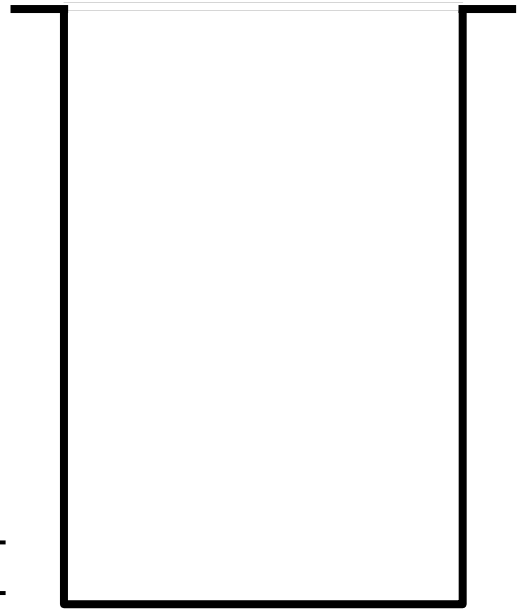
Heap and stack memory

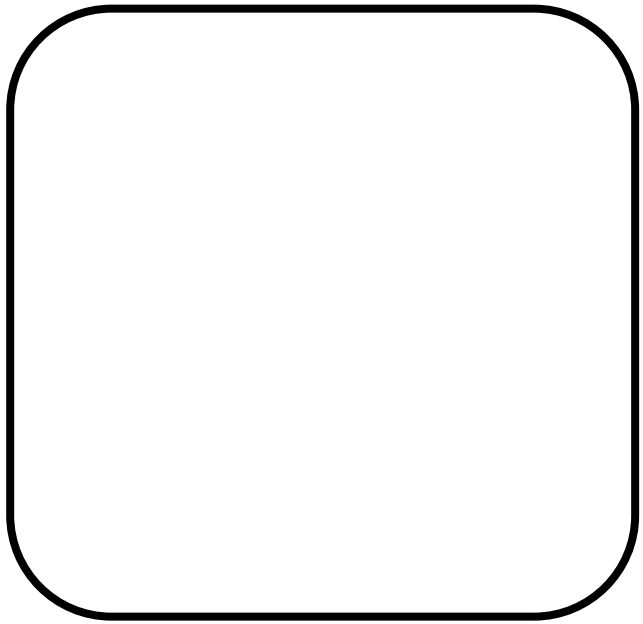




```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
       delete z;
       z = 0;
   }
   return 0;
}
```

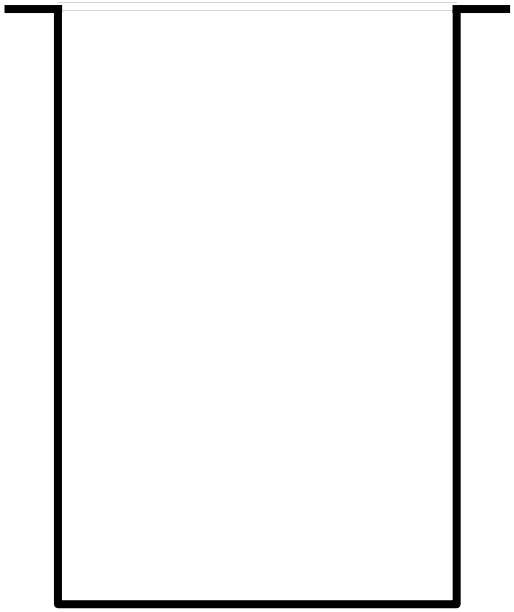
Memory leak!

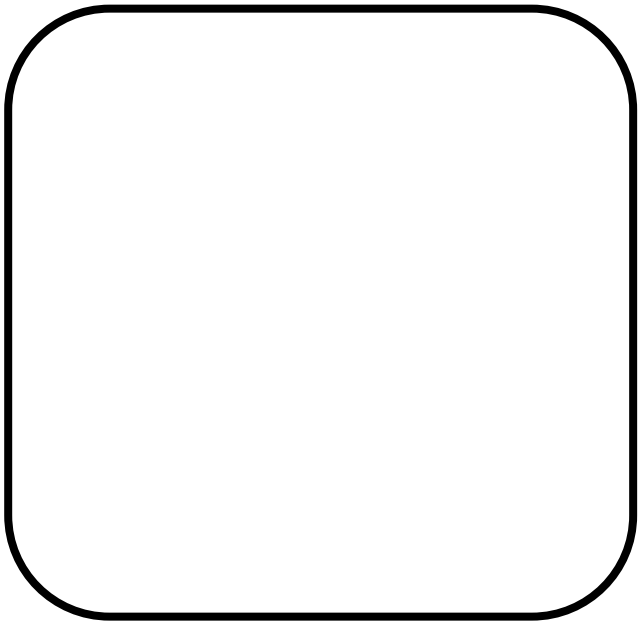




```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
       delete z;
       z = 0;
   }
   return 0;
}
```

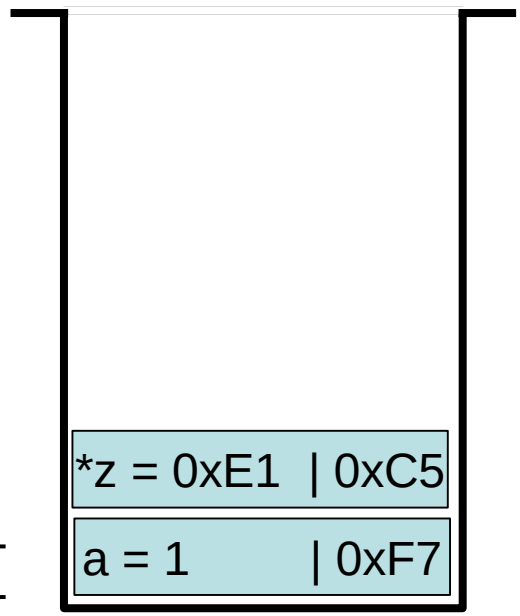
Memory leak!

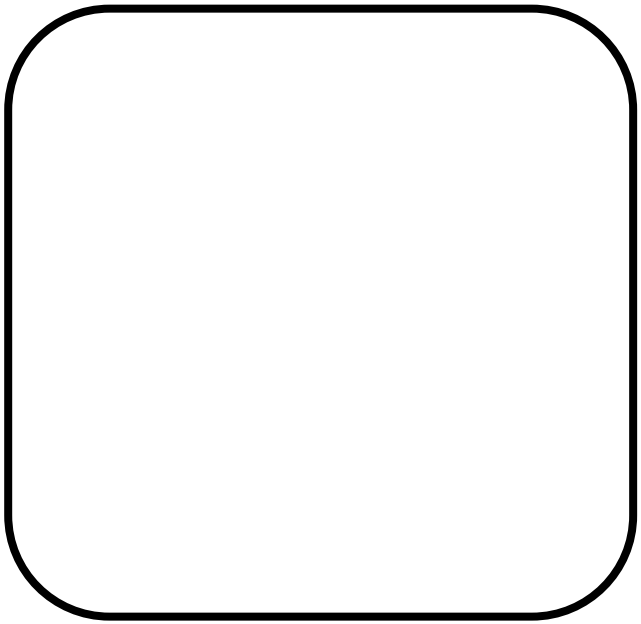




```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```

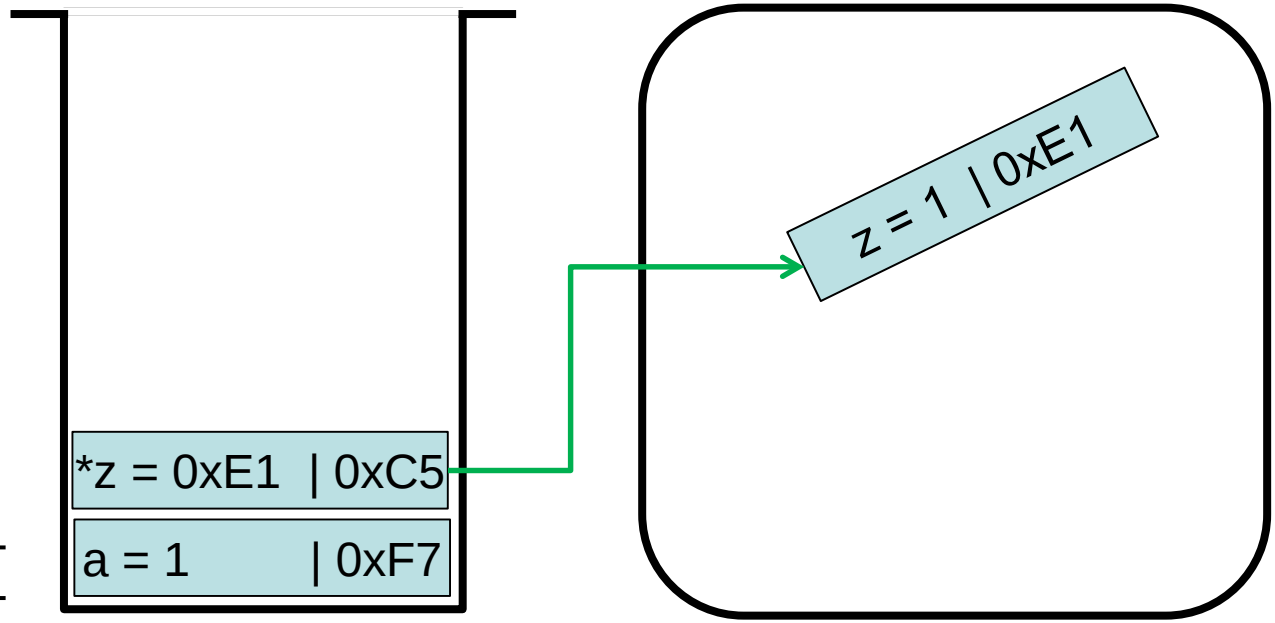
Memory leak!





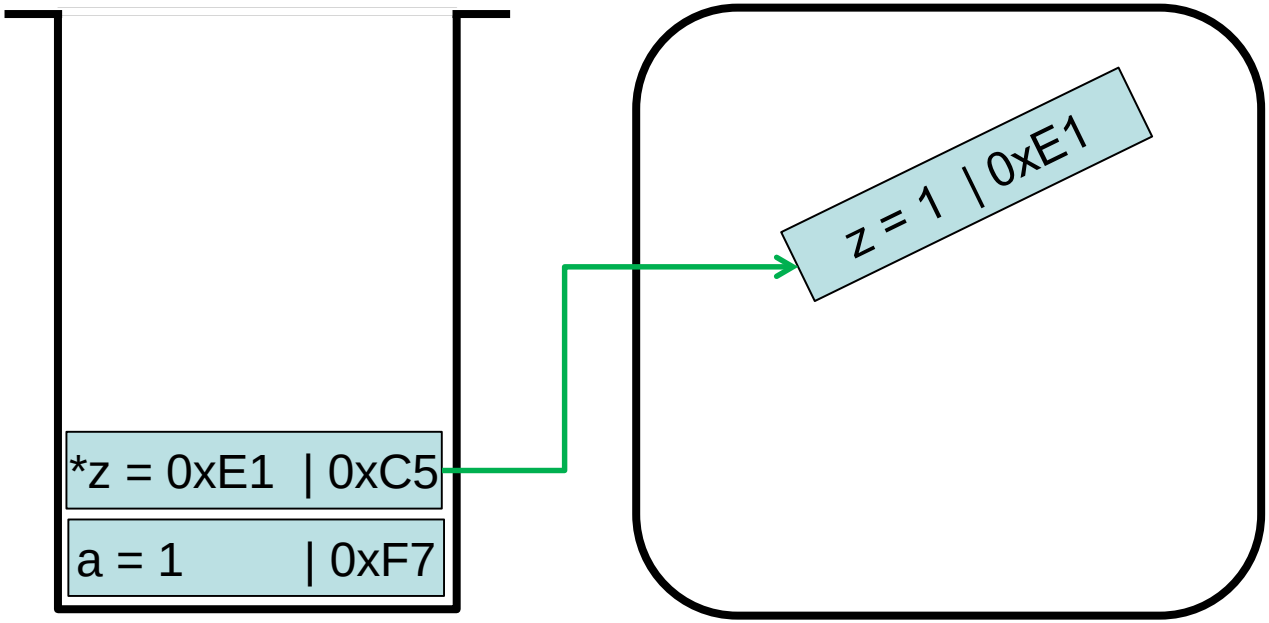
```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```

Memory leak!



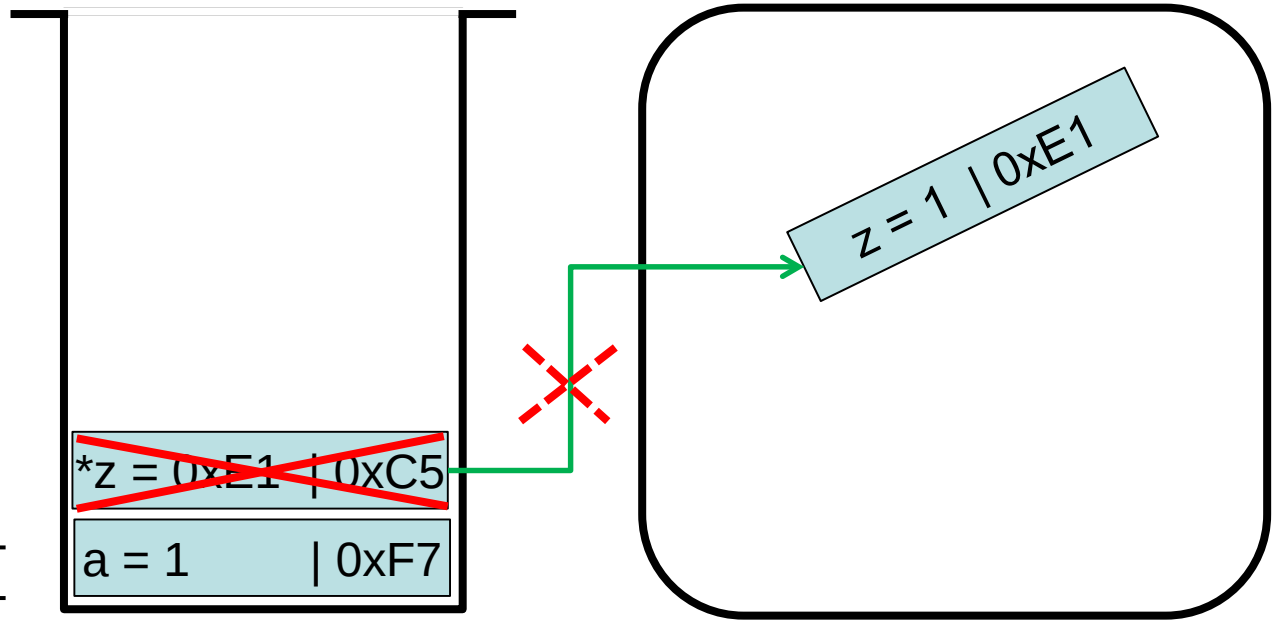

```
int main(){
   int a = 1;
   if ( a > 0 ){
      int *z = new int(1);
      // do something
   }
   return 0;
}
```

Memory leak!



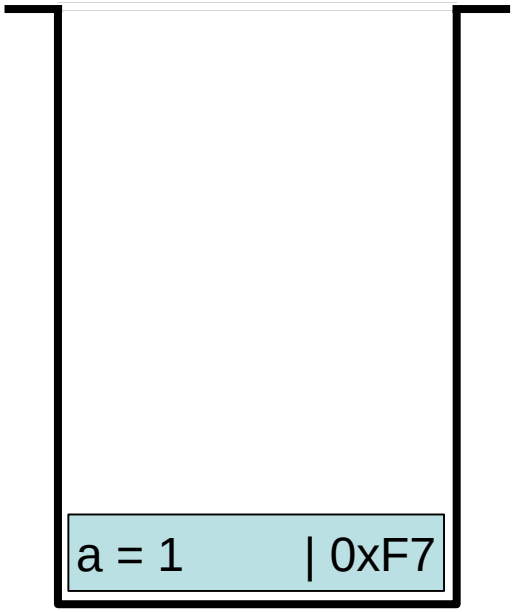

```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```

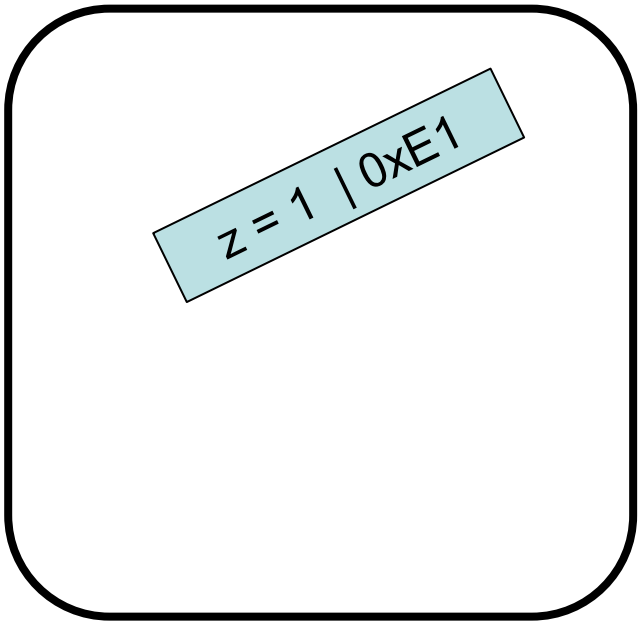
Memory leak!




```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```

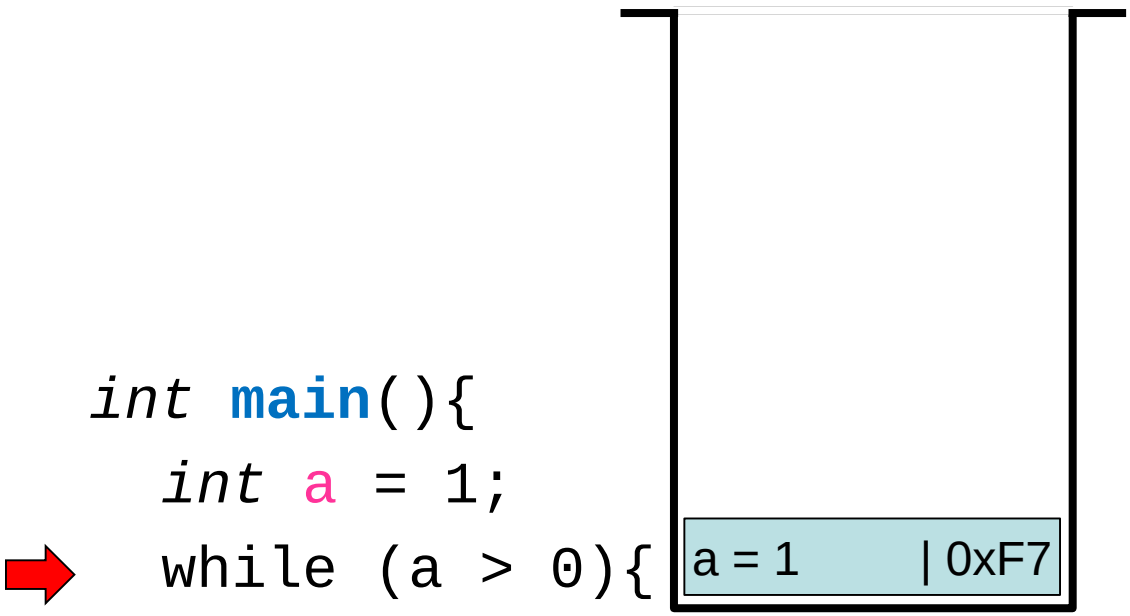
Memory leak!

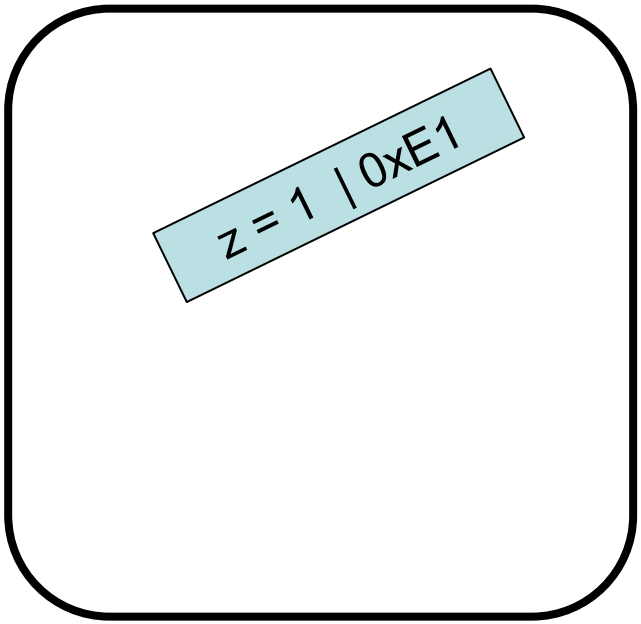





```
int main(){
   int a = 1;
   if ( a > 0 ){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```

Memory leak!

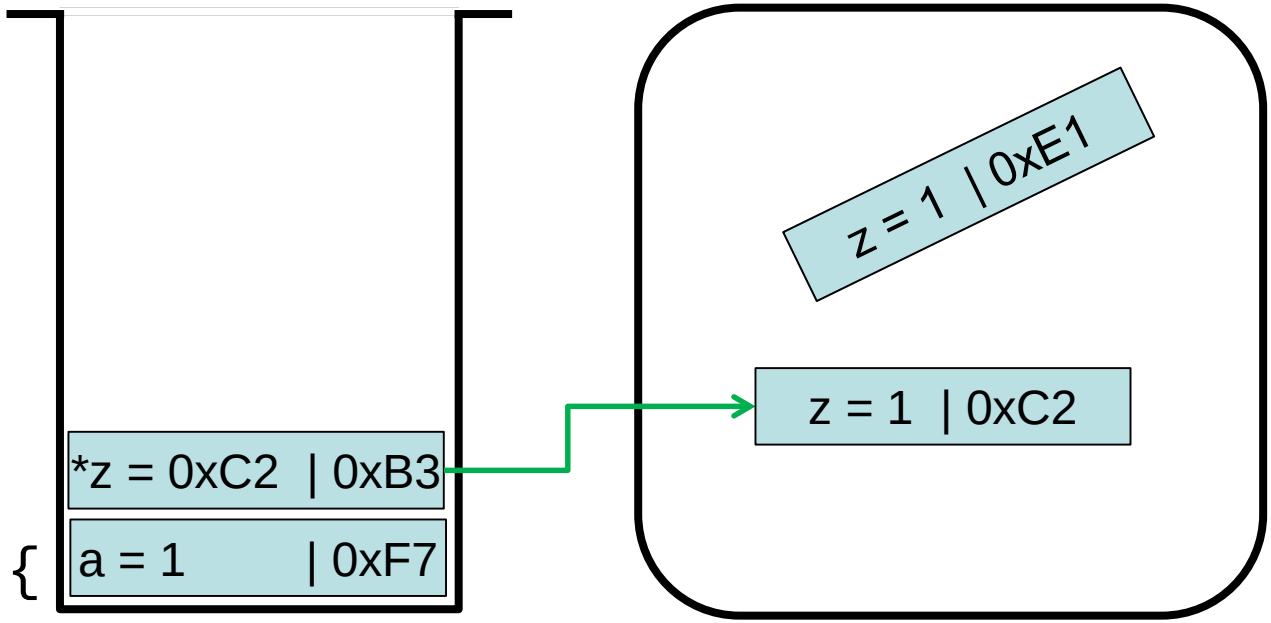





```
int main(){
   int a = 1;
   while (a > 0){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```

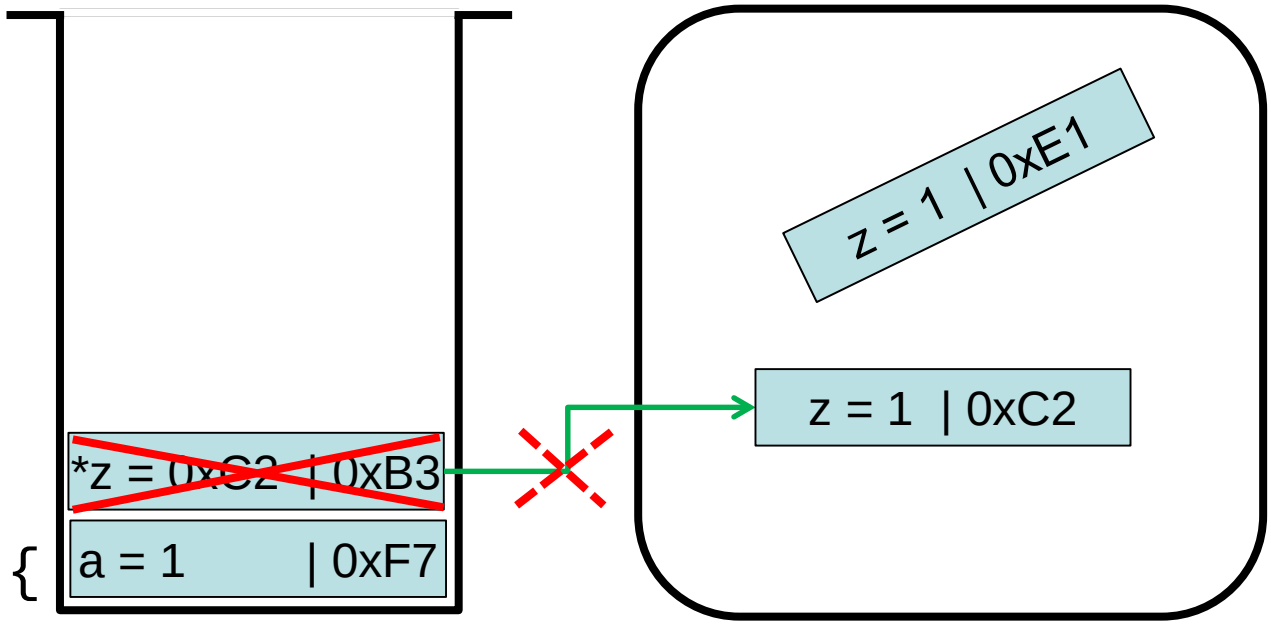
Memory leak!

```
int main(){
   int a = 1;
   while (a > 0){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```



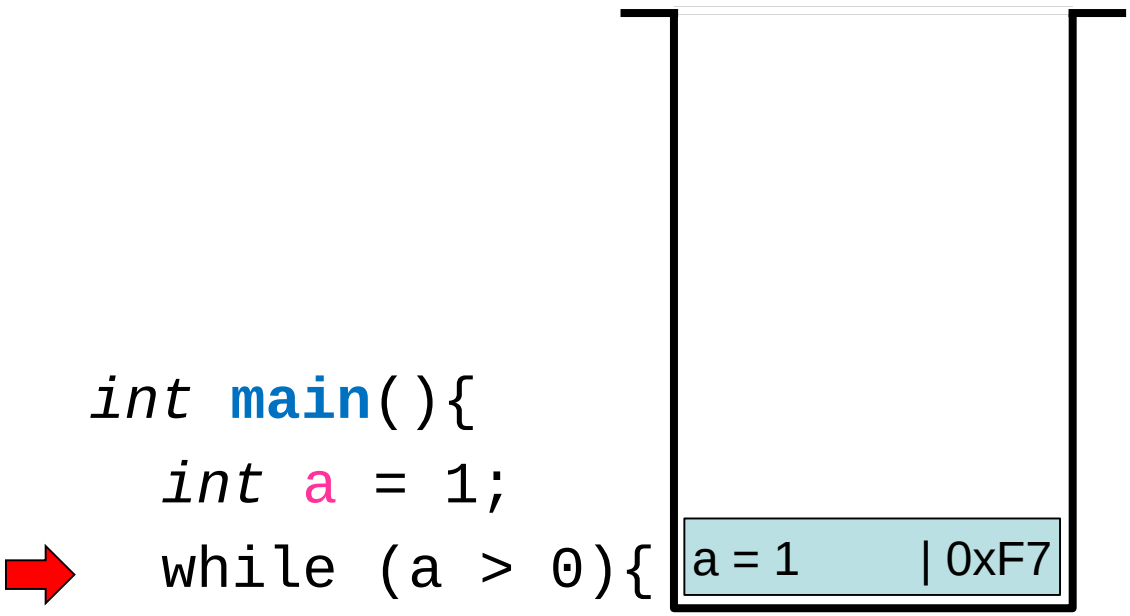
Memory leak!

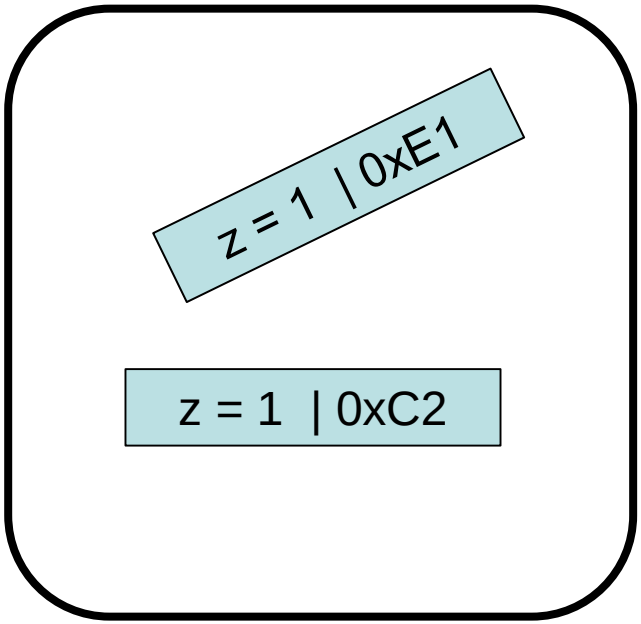
```
int main(){
   int a = 1;
   while (a > 0){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```



Memory leak!

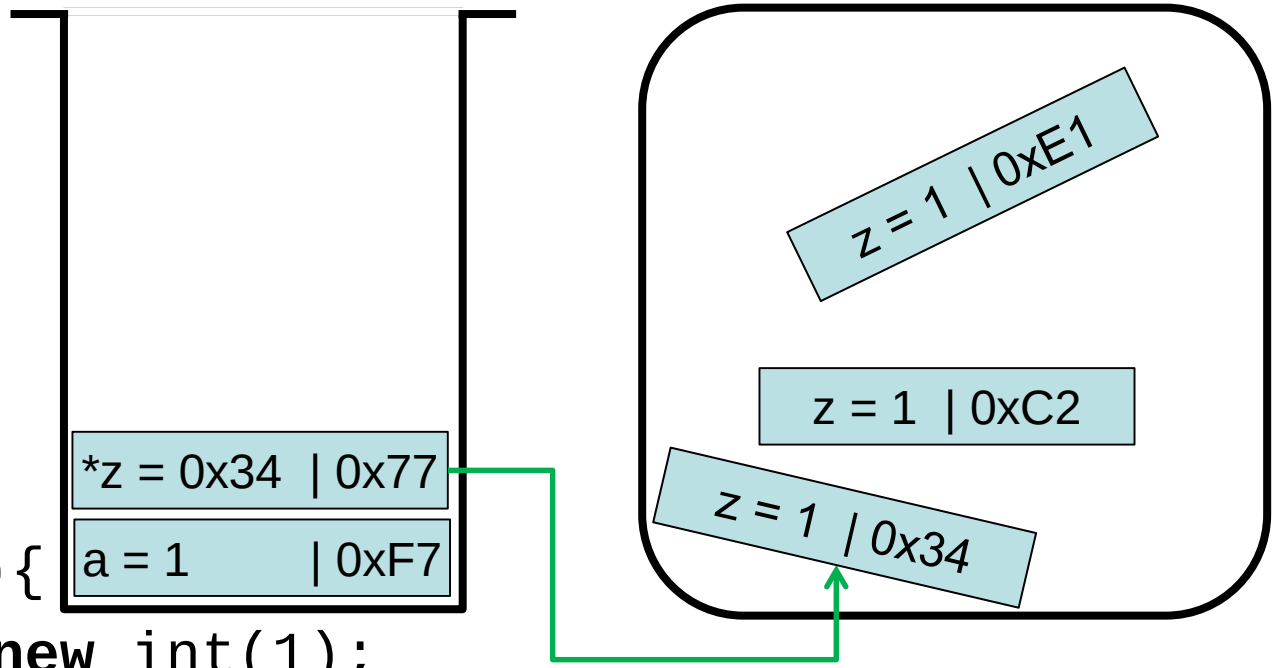
```
int main(){
   int a = 1;
   while (a > 0){
      int *z = new int(1);
      // do something
   }
   return 0;
}
```





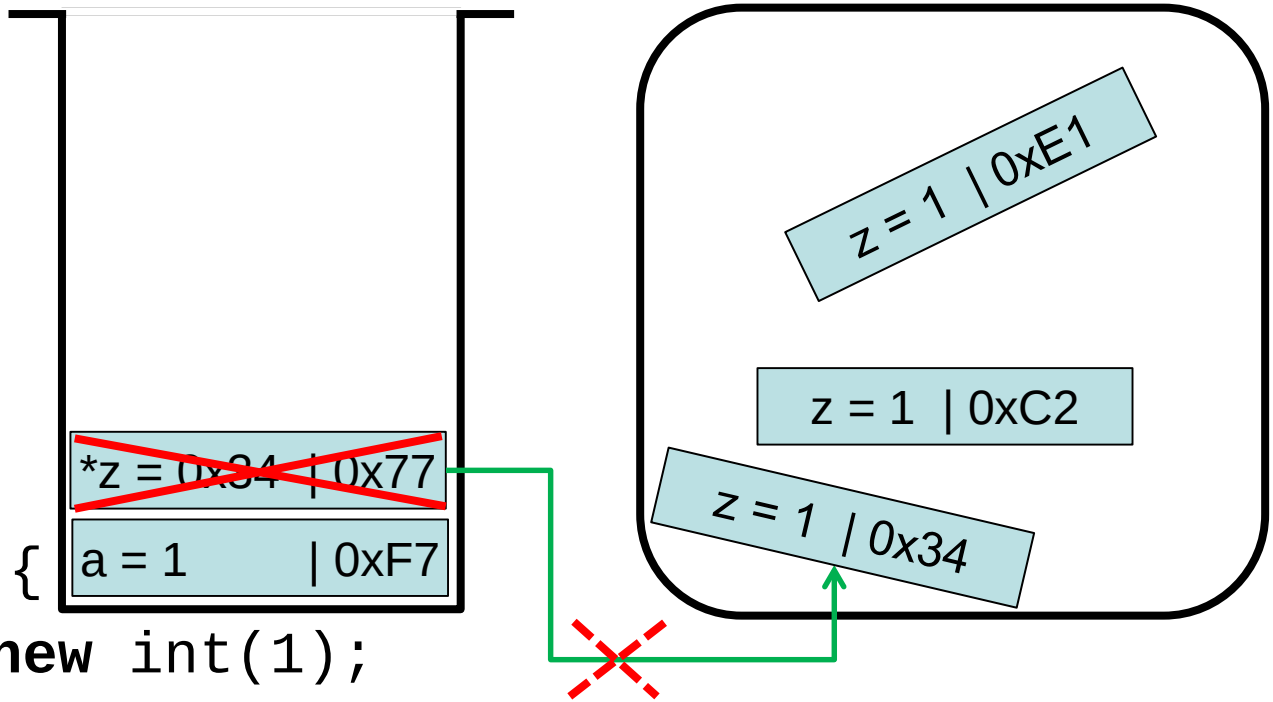
Memory leak!

```
int main(){
   int a = 1;
   while (a > 0){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```



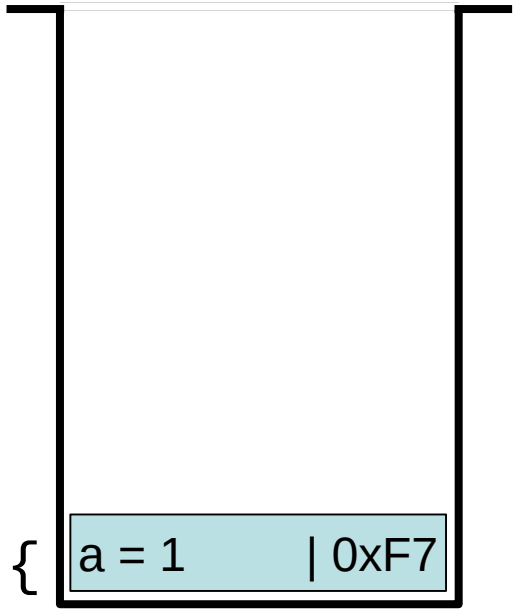
Memory leak!

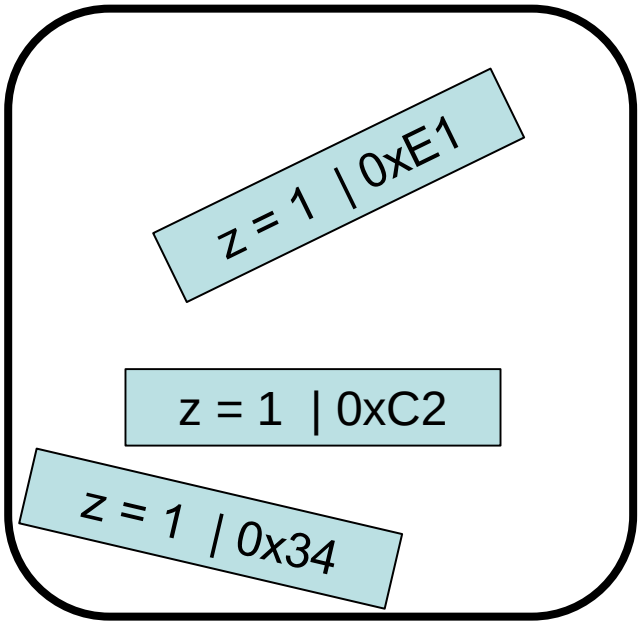
```
int main(){
   int a = 1;
   while (a > 0){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```



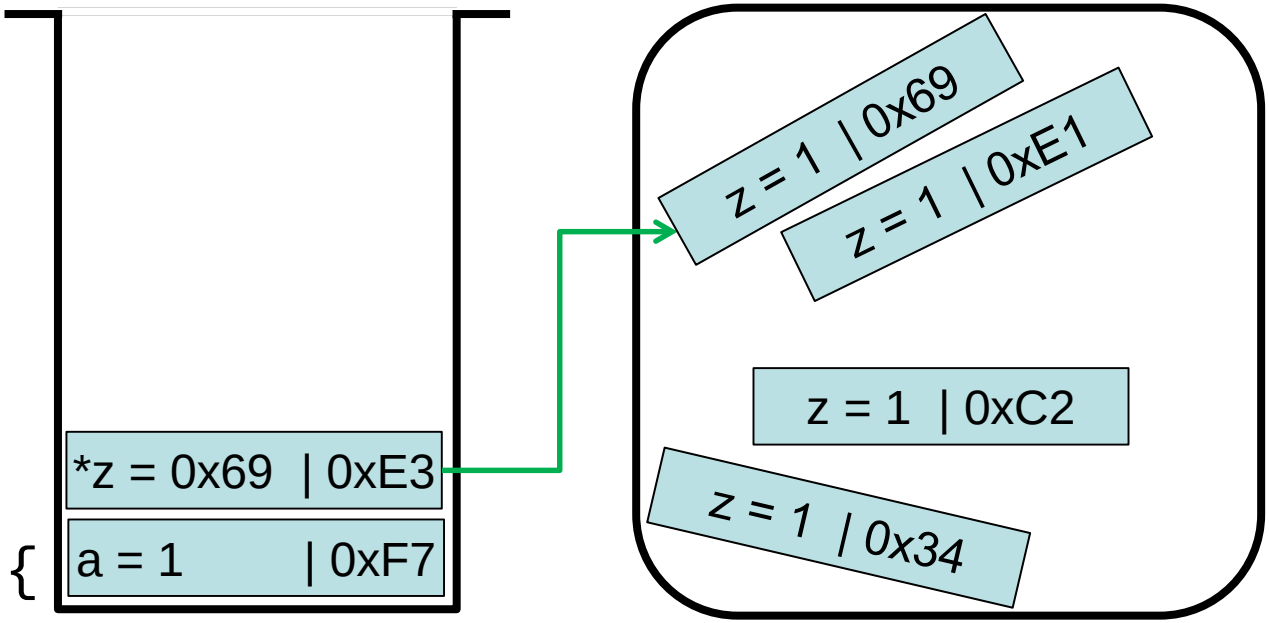
Memory leak!

```
int main(){
   int a = 1;
   while (a > 0){
      int *z = new int(1);
      // do something
   }
   return 0;
}
```





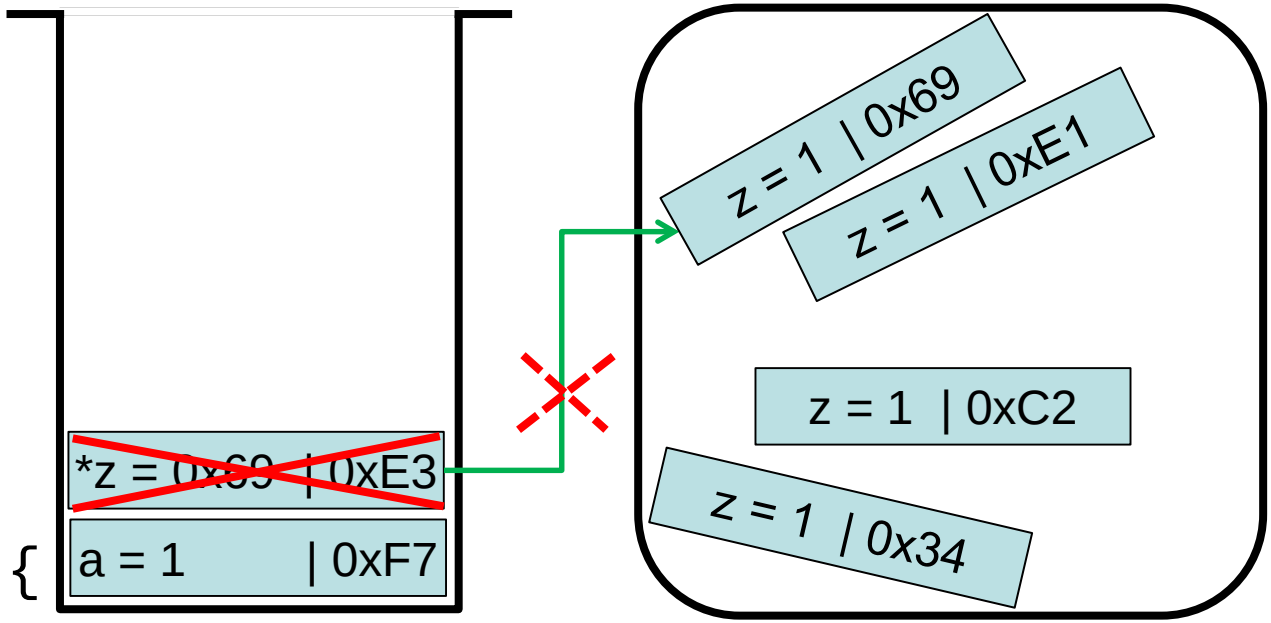
Memory leak!




```
int main(){
   int a = 1;
   while (a > 0){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```

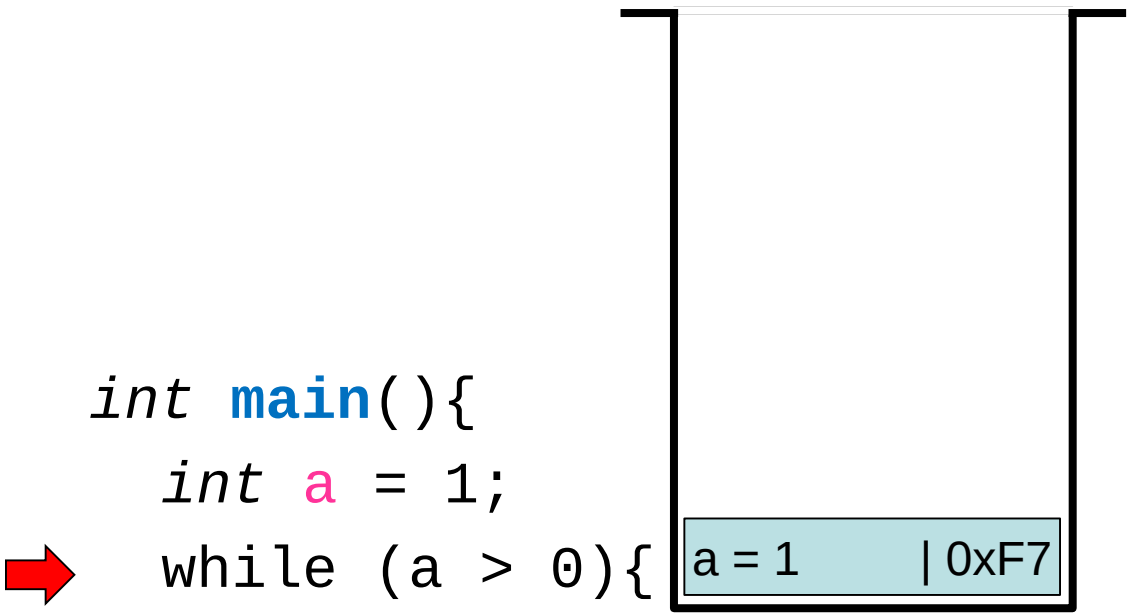
Memory leak!

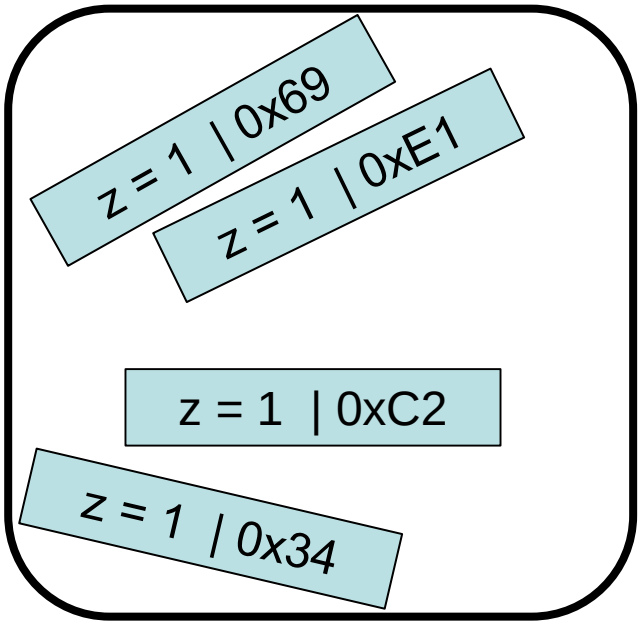
```
int main(){
   int a = 1;
   while (a > 0){
       int *z = new int(1);
       // do something
   }
   return 0;
}
```



Memory leak!

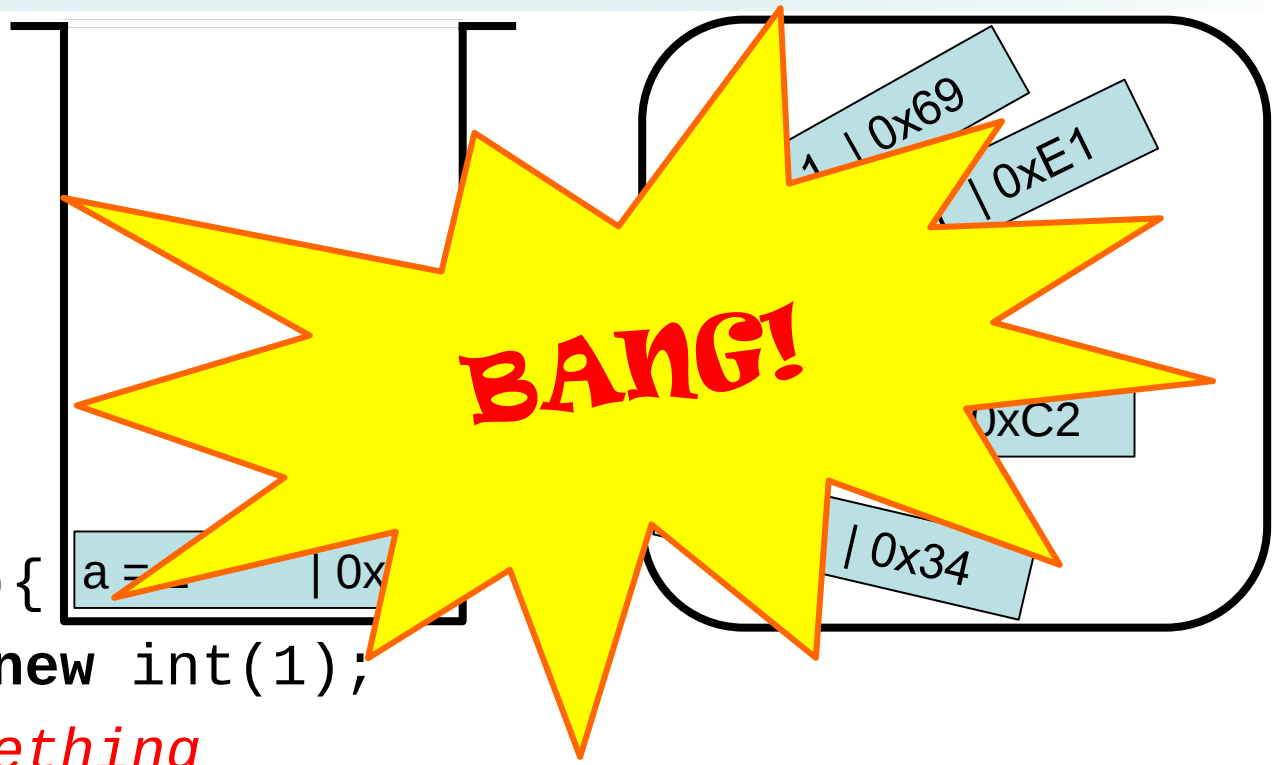
```
int main(){
   int a = 1;
   while (a > 0){
      int *z = new int(1);
      // do something
   }
   return 0;
}
```





Memory leak!

```
int main(){
   int a = 1;
   while (a > 0){
      int *z = new int(1);
      // do something
   }
   return 0;
}
```



Arrays, operator [ ]

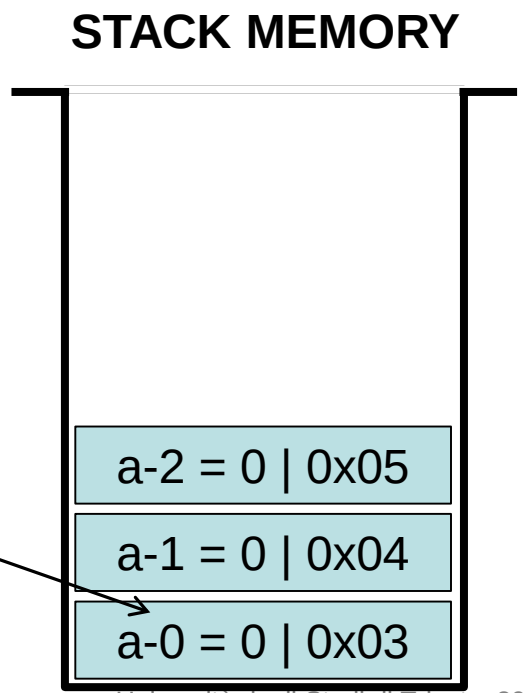
Array: is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier

For example, we can store 3 values of type int in an array without having to declare 3 different variables, each one with a different identifier. Instead of that, using an array we can store 3 different values of the same type, int for example, with a unique identifier:

```
int a[3];
```

THE FIRST INDEX IS ALWAYS ZERO!

(unlike FORTRAN)

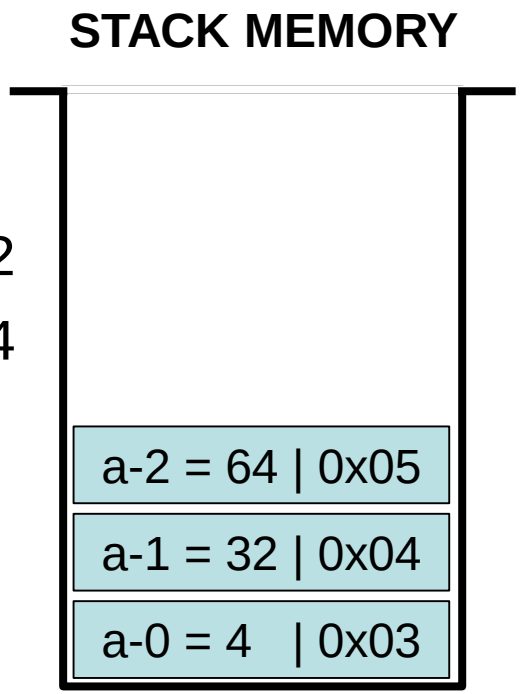


Arrays, operator [ ]

Array: is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier

Initialization at declaration can be done using `{...,...,...}`, individual elements are accessed via “[index]”:

```
int a[3] = {4,32,64};
cout << a[0] << “\n”; // output is 4
cout << a[1] << “\n”; // output is 32
cout << a[2] << “\n”; // output is 64
a[1] = 7;
cout << a[1] << “\n”; // output is 7
```

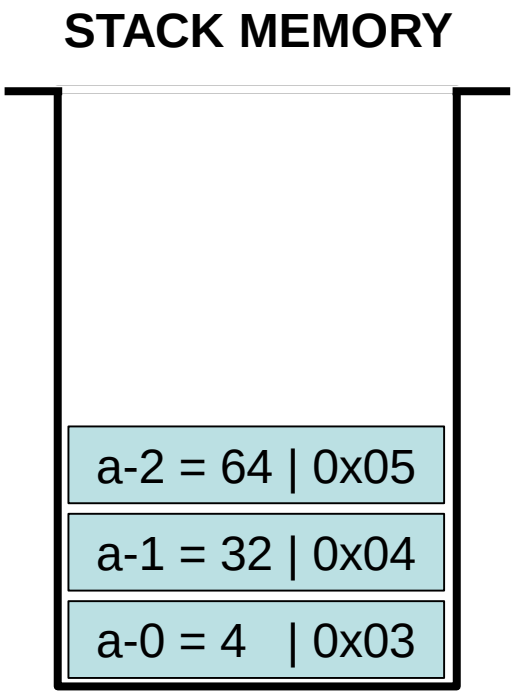


Arrays, operator [ ]

Array: is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier

When initialized, dimension can be omitted:

```
int a[] = {4,32,64};
```

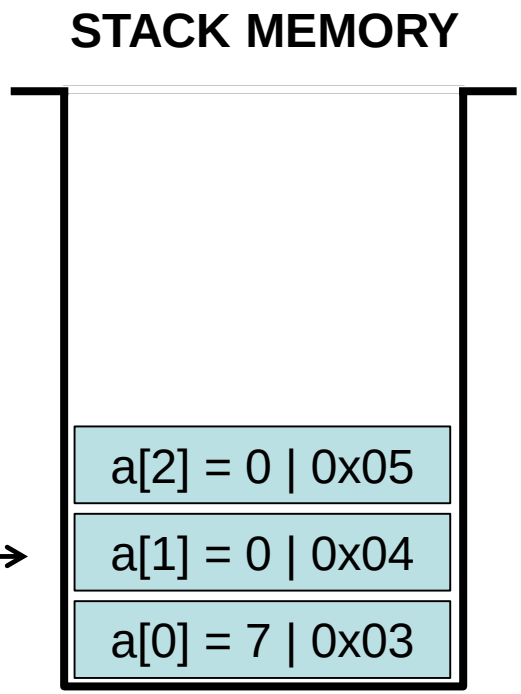


Arrays, operator [ ]

Array: is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier

WARNING (common mistake): you cannot initialize all the array with the same value using the syntax “{value}”:

```
int a[3] = {7};
cout << a[0] << “\n”; // output is 7
cout << a[1] << “\n”; // output is 0
cout << a[2] << “\n”; // output is 0
```



use memset or a loop:

```
memset(a,7,3*sizeof(int));
for (int i=0; i<3; i++) a[i] = 7;
```

Arrays, operator [ ]

Array: is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier

Arrays can be multi-dimensional (a matrix):

```
      row column
      ↘   ↘
int b[2][2];
```

$$b = \begin{Bmatrix} 0 & 1 \\ 2 & 3 \end{Bmatrix}$$

and are initialized row by row, column by column:

```
int b[2][2] = {0,1,2,3}; // bad formatting
int b[2][2] = {0,1,
               2,3}; // much better
cout << b[0][0] << "\n"; // 0
cout << b[0][1] << "\n"; // 1
cout << b[1][0] << "\n"; // 2
cout << b[1][1] << "\n"; // 3
```

C++ and strings

C-style strings:

```
const char *name; // name is a pointer to a
  character constant (??)
char *const name; // name is constant pointer to
  character string (string can be changed!)
name = “Emiliano”;
cout << “ name is ” << name << “\n”;
```

C++-style strings:

```
#include <string>
...
string name;
name = “Emiliano”;
cout << “ name is ” << name.c_str() << “\n”;
```

http://www.cplusplus.com/reference/string/string/

C++ and strings

strings.cpp:

```
#include <iostream>
#include <string>
using namespace std;
int main(){
  const char *myString1 = "C-style string";
  cout << " myString1 is \"" << myString1 << "\"\n";
//  char *const myString2 = " string two "; // deprecated
//  cout << " myString2 is \"" << myString2 << "\"\n";

  string myString3 = "C++ string";
  cout << " myString3 is \"" << myString3.c_str() << "\"\n";

  string name;
  name = "Emiliano";
  cout << " name is \"" << name.c_str() << "\"\n";
  cout << " size of name is " << name.size() << "\n";

  string *surname = new string();
  surname->assign("Mocchiutti");
  cout << " surname is \"" << surname->c_str() << "\"\n";
  cout << " size of surname is " << surname->size() << "\n";
  delete surname;
  return 0;
}
```

Reading inputs from prompt

```
#include <iostream>
using namespace std;
int main(){
   cout “ hello ” << endl;
   return 0;
}
```

```
bash> ./hello
 hello
bash>
```

Reading inputs from prompt

```
bash> ./hello ciao 1.1 b
 ciao
 value: 1.1
 string: b
bash>
```

Reading inputs from prompt

```
#include <stdlib.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
```

```
bash> ./hello ciao 1.1 b
 ciao
 value: 1.1
 string: b
bash>
```

array of strings that form the command line (executable included!)

number of inputs on the command line (executable included!)

Reading inputs from prompt

```
#include <stdlib.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
```

```
bash> ./hello ciao 1.1 b
 ciao
value: 1.1
 string: b
bash>
```

array of strings that form the command line (executable included!)

number of inputs on the command line (executable included!)

bash> ./hello ciao 1.1 b
Number of inputs: 4 hence:
argc = 4 argv[4]={“./hello”, “ciao”, “1.1”, “b”}

Reading inputs from prompt

```
#include <stdlib.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
```

```
bash> ./hello ciao 1.1 b
 ciao
 value: 1.1
 string: b
bash>
```

array of strings that form the command line (executable included!)

number of inputs on the command line (executable included!)

bash> ./hello ciao 1.1 b
Number of inputs: 4 hence:

numbers as strings!!

argc = 4 argv[4]={“./hello”, “ciao”, “1.1”, “b”}

Reading inputs from prompt

```
#include <stdlib.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
  // how to use inputs
  cout argv[1] << endl;
  cout “value: ” << argv[2] << endl;
  cout “string: ” << argv[3] << endl;
  float myFloat = atof(argv[2]);
  ... (if needed do something else...)
  return 0;
}
```

```
bash> ./hello ciao 1.1 b
 ciao
value: 1.1
 string: b
bash>
```

Reading inputs from prompt

```
#include <stdlib.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
  // how to use inputs
  cout argv[1] << endl;
  cout “value: ” << argv[2] << endl;
  cout “string: ” << argv[3] << endl;
  float myFloat = atof(argv[2]);
  ... (if needed do something else...)
  return 0;
}
```

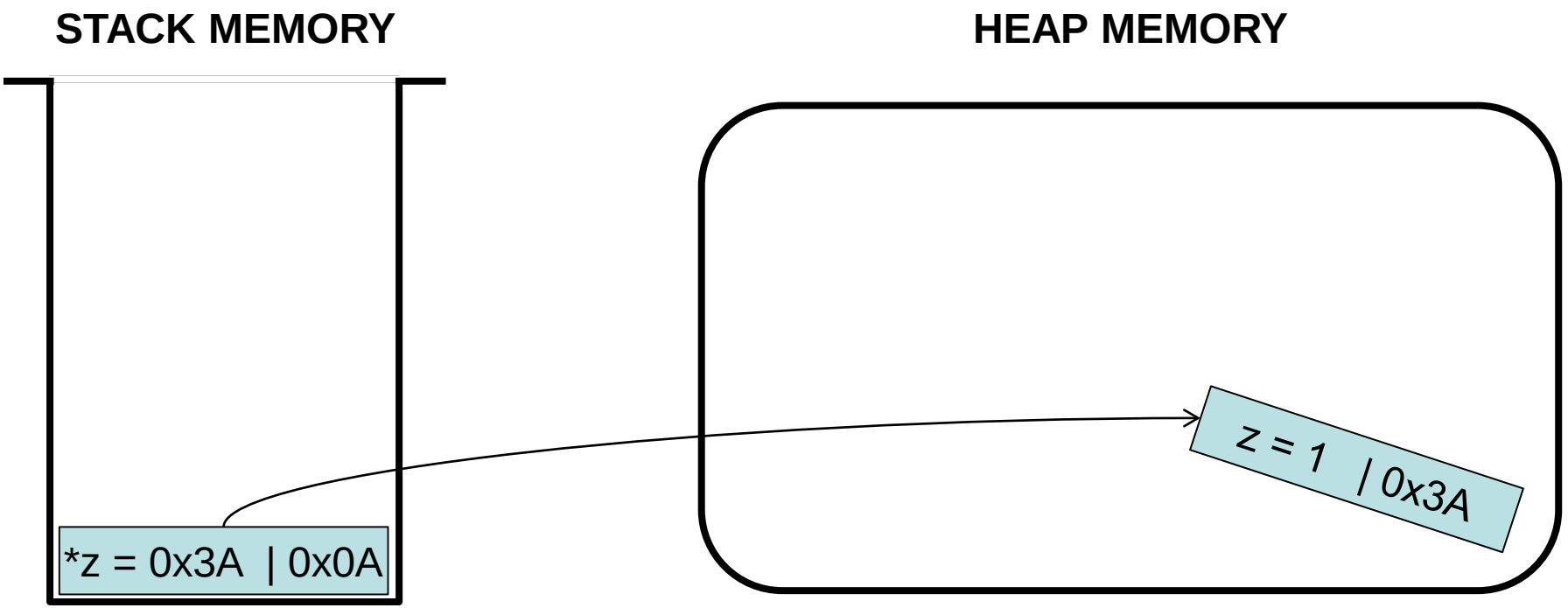
```
bash> ./hello ciao 1.1 b
 ciao
 value: 1.1
 string: b
bash>
```

it converts strings to floats
look also for “atoi”

http://www.cplusplus.com/reference/cstdlib/atof/
http://www.cplusplus.com/reference/cstdlib/atoi/

Pointers

Pointer: a variable which contains the address of a memory region where a certain object is stored




```
int *z = new int(1);
```

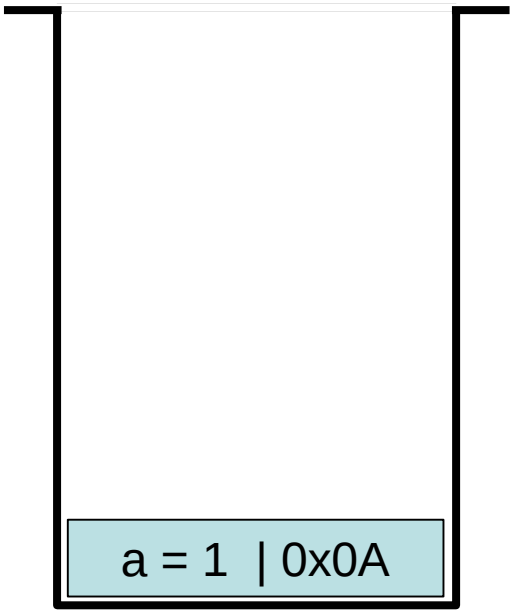
References: “&”

Reference: an alias to an existing object (usually a variable)

References allow you to create a second name for the a variable that you can use to read or modify the original data stored in that variable.

```
int a = 1;
```

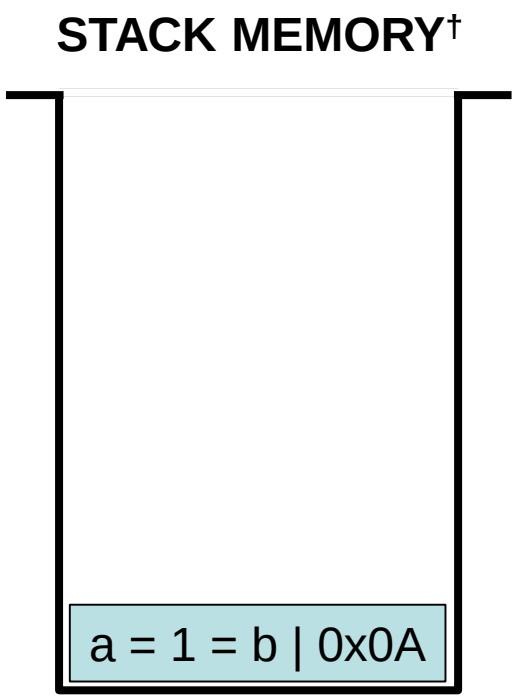
STACK MEMORY



References: “&”

Reference: an alias to an existing object (usually a variable)

References allow you to create a second name for the a variable that you can use to read or modify the original data stored in that variable.



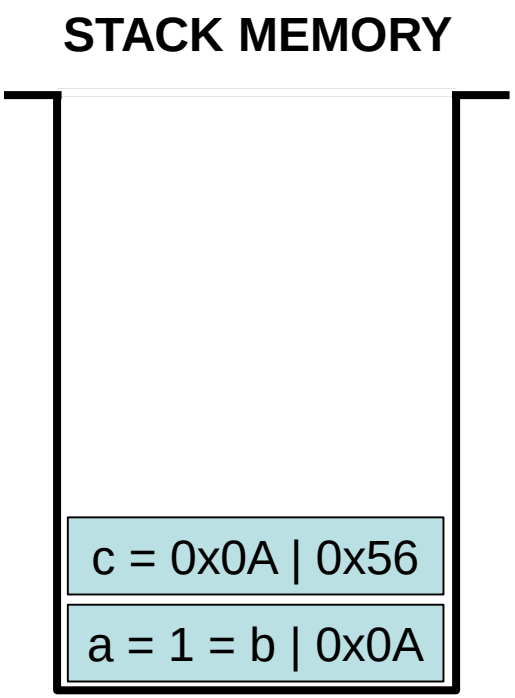

```
int a = 1;
int& b = a; // or int &b = a;
b = 64;
cout << a << “\n”; // prints “64”
```

†WARNING: **not** a realistic representation of what happens in memory!! very simplified vision!

References: “&”

Reference: an alias to an existing object (usually a variable)

References allow you to create a second name for the a variable that you can use to read or modify the original data stored in that variable.




```
int a = 1;
int& b = a; // or int &b = a;
b = 64;
cout << a << “\n”; // prints “64”
```

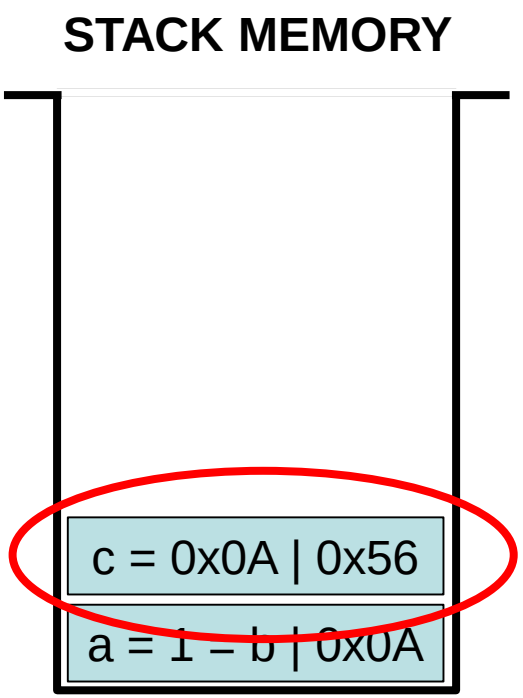
“&” ampersand should be read “address-of”

```
int &b = a; // the address-of b is
            the same of a
int c = &a; // put in the variable
      called c the address of a
```

References: “&”

Reference: an alias to an existing object (usually a variable)

References allow you to create a second name for the a variable that you can use to read or modify the original data stored in that variable.




```
int a = 1;
int& b = a; // or int &b = a;
b = 64;
cout << a << “\n”; // prints “64”
```

“&” ampersand should be read “address-of”

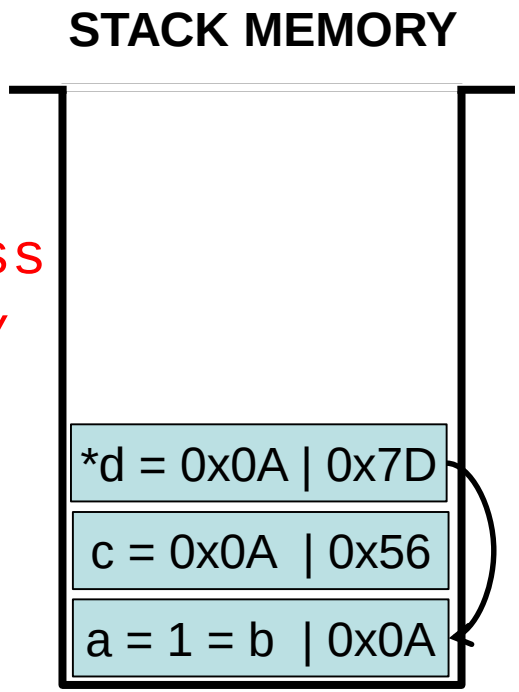
```
int &b = a; // the address-of b is
            the same of a
int c = &a; // put in the variable
      called c the address of a
```

Pointers

Pointer: a variable which contains the address of a memory region where a certain object is stored

```
int a = 1;          // a is one
int &b = a;         // b is a
int c = &a; //not allowed, meaningless
int c = reinterpret_cast<int>(&a); //
 c takes the address of a; that is
 c contains the value 0x0A = 10

int *d = &a;        // d points to a
```



the pointer can point to any position in memory, not only heap but also stack memory! take a look at “pointers.cpp”

How to refer to objects

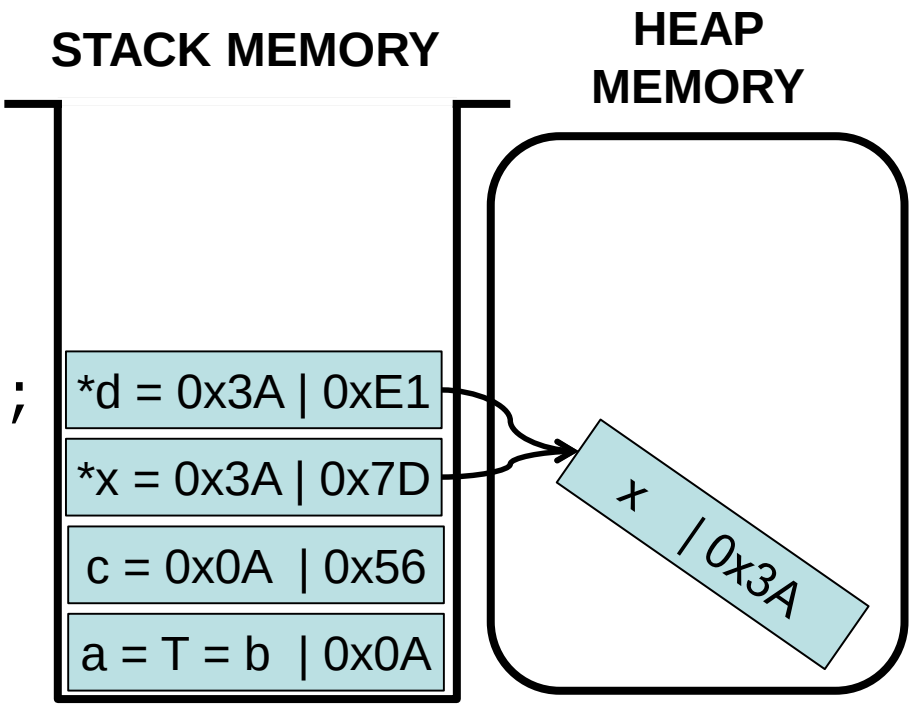
Hypotesis: object of type T has a certain built-in function called "doSomething()"

Stack object, use dot "." : `T.doSomething();`

Heap object, use *arrow* "->" : `T->doSomething();`

```
T a(initParameters);
a.doSomething();
T &b = a;
b.doSomething();


T *x = new T(initParameters);
x->doSomething();
T *d = x;// copy into d the
         content of *x
d->doSomething();
```



More on functions

- Functions declaration and implementation
- Passing arguments to functions, data hiding!
- Pointers and references as function arguments

More on functions

Functions are pieces of code performing a specific task.

Function declaration (like `int a;`):

```
int countStudents(int, int, float);
```

Function implementation (like `a=1;`):

```
int countStudents(int nStudents, int nMissing, float
   missingProbability){
   int nStudentsHere;
   // [...] block performing the task
   return nStudentsHere;
}
```

More on functions

Functions must be or declared or directly implemented BEFORE their use, example 1:

```
int countStudents(int, int, float);
int main(){
   // do something
   int nStudents = countStudents(1,1,0.);
   // etc. etc.
   return 0;
}
int countStudents(int nStudents, int nMissing, float
   missingProbability){
   int nStudentsHere;
   // [...] block performing the task
   return nStudentsHere;
}
```

More on functions

Functions must be or declared or directly implemented BEFORE their use, example 2:

```
int countStudents(int nStudents, int nMissing, float
  missingProbability){
  int nStudentsHere;
  // [...] block performing the task
  return nStudentsHere;
}
int main(){
  // do something
  int nStudents = countStudents(1,1,0.);
  // etc. etc.
  return 0;
}
```

Functions, data hiding!

Programs variables are intrinsecally protected by C++, data passed as arguments to functions are COPIED:

```
void increment(int m){ // alias for int m = a;
   m++;
   cout << " m is " << m << "\n";
}

int main(){
   int a = 0;
   cout << " a is " << a << "\n"; // output: a is 0
   increment(a);                 // output: m is 1
   cout << " a is " << a << "\n"; // output: a is 0
   return 0;
}
```

Functions, * and & as arguments

Programs variables are intrinsecally protected by C++, data can be modified if a reference is passed:

```
void increment(int &m){ // alias int &m = a; address of m is
   the address of a
   m++;
   cout << " m is " << m << "\n";
}

int main(){
   int a = 0;
   cout << " a is " << a << "\n"; // output: a is 0
   increment(a);                 // output: m is 1
   cout << " a is " << a << "\n"; // output: a is 1
   return 0;
}
```

Functions, * and & as arguments

Programs variables are intrinsecally protected by C++, data can be modified if a reference to a pointer is passed (NOT suggested as a solution):

```
void increment(int *m){ // alias int *m = &a; address of m is
   the address of a
   (*m)++;
   cout << " m is " << *m << "\n";
}

int main(){
   int a = 0;
   cout << " a is " << a << "\n"; // output: a is 0
   increment(&a);                // output: m is 1
   cout << " a is " << a << "\n"; // output: a is 1
   return 0;
}
```

Functions, * and & as arguments

Programs variables are intrinsecally protected by C++, data can be modified if pointer to pointer is passed (very common):

```
void increment(int *m){ // alias int *m = a; address pointed by
   m is the address pointed by a
   (*m)++;
   cout << " m is " << *m << "\n";
}

int main(){
   int *a = new int(0);
   cout << " a is " << *a << "\n"; // output: a is 0
   increment(a);                  // output: m is 1
   cout << " a is " << *a << "\n"; // output: a is 1
   return 0;
}
```

2017/03/10 – take home message

- expressions, operators, conditional and iteration statements, type… very similar to FORTRAN
- variable declaration is mandatory in C++
- do not be scared by pointers…
- heap and stack memory, simplified view
- read references (&) “address-of”
- passing arguments to functions:
  - variables are usually copied!
  - pointers should be used to change variables inside functions
- read declarations from right to left
- in case of doubts look at www.learncpp.com , e.g. the “main” arguments explanation can be found here: http://www.learncpp.com/cpp-tutorial/713-command-line-arguments/

Examples and exercises

- Download, look, compile, and run:

hello.cpp, typesSize.cpp, pointers.cpp, moreFunctions.cpp, moreFunctions2.cpp, moreFunctions3.cpp, moreFunctionsWrong.cpp, constDeclarations.cpp, strings.cpp

- write a program which calculates the area of a given trapezoid and print the result on the STDOUT
- change it in order to accept as input from the command line three numbers - minor basis, major basis and height of a trapezoid

Exercises - extra

- update the previous program in order to be able to specify on the command line the type of input, i.e. it should be possible to run it as:

```
bash> ./CalculateArea --majorBasis 1. --minorBasis 0.3 --height 5.
```

or

```
bash> ./CalculateArea --minorBasis 0.3 --height 5. --majorBasis 1.
```

or any other permutation of inputs.