Agenda

- The main body and cout
- Fundamental data types
- Declarations and definitions
- Control structures
- References, pass-by-value vs pass-by-references

Memory regions

EACH C++ VARIABLE IS STORED IN A MEMORY REGION

THE SIZE OF THE REGION DEPENDS ON THE VARIABLE'S TYPE

Sizes of various types


```
cout << "sizeof(char)      = " << sizeof(char) << endl;
cout << "sizeof(char&)     = " << sizeof(char&) << endl;
cout << "sizeof(int)       = " << sizeof(int) << endl;
cout << "sizeof(int&)      = " << sizeof(int&) << endl;
cout << "sizeof(long int)  = " << sizeof(long int) << endl;
cout << "sizeof(bool)      = " << sizeof(bool) << endl;
cout << "sizeof(float)     = " << sizeof(float) << endl;
cout << "sizeof(double)    = " << sizeof(double) << endl;
cout << "sizeof(string)    = " << sizeof(string) << endl;
```

Where in memory? Use “address of” op


```
#include <iostream>
using namespace std;

int main() {
    int a = 12345;
    cout << "Address of a is at: " << &a << endl; // something like 0x7fff6425d7c4
    return 0;
}
```

32-bit architecture: an address is 4 bytes long
64-bit architecture: an address is 8 bytes long

Pointers

- A POINTER IS A VARIABLE THAT HOLDS A MEMORY ADDRESS
- WE OFTEN WANT A POINTER TO A PARTICULAR TYPE
- POINTERS ARE EXTREMELY POWERFUL (JAVA HIDES IT FROM US!)

Declaring pointers


```
int *i_ptr;     // i_ptr is a pointer to an integer
char *c_ptr;    // c_ptr is a pointer to a character
string *s_ptr; // s_ptr is a pointer to a string
```

```
cout << "sizeof(i_ptr)     = " << sizeof(char*) << endl;
cout << "sizeof(char*)     = " << sizeof(char*) << endl;
cout << "sizeof(int*)      = " << sizeof(int*) << endl;
cout << "sizeof(string*)   = " << sizeof(string*) << endl;
```

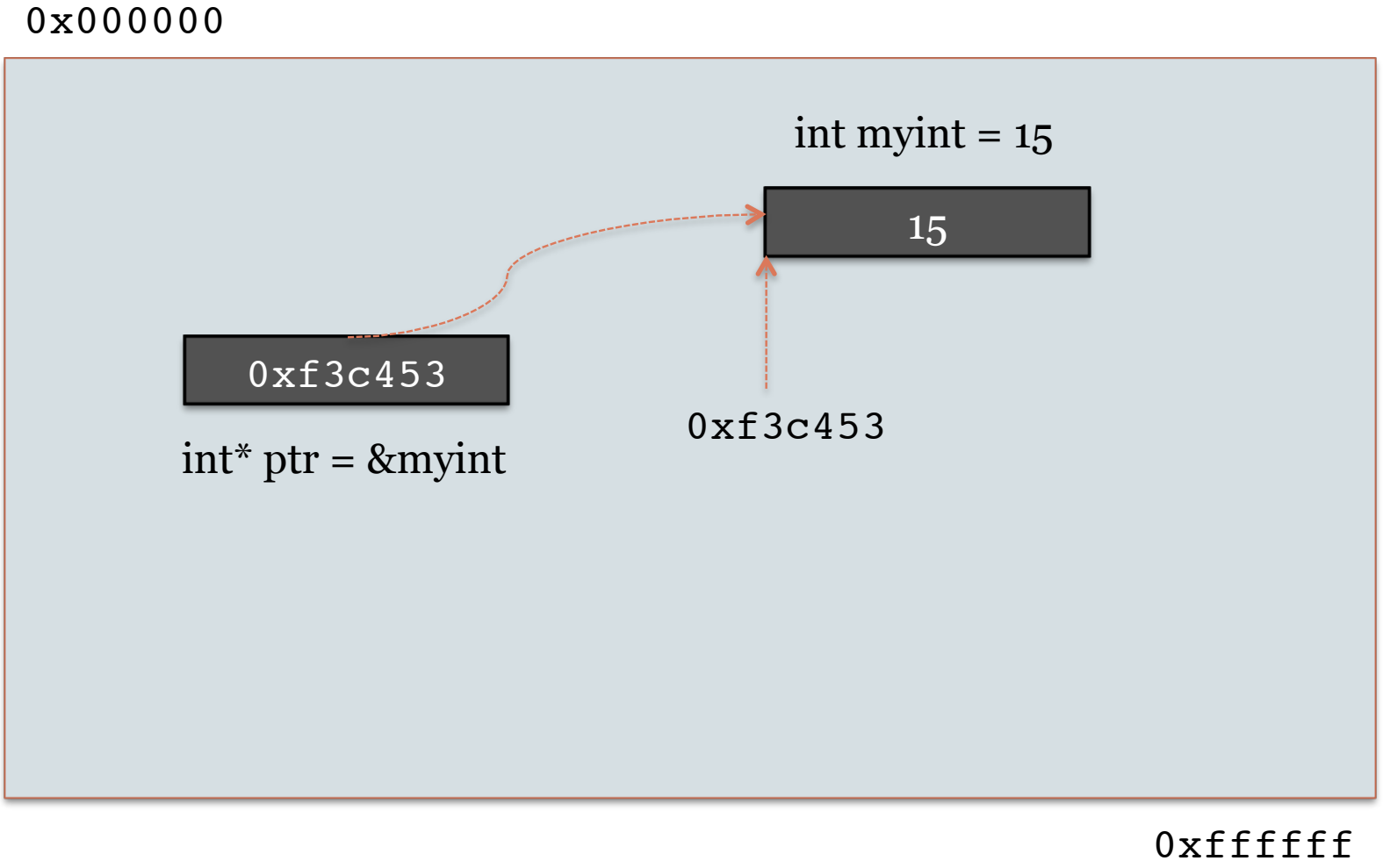
Assigning and dereferencing

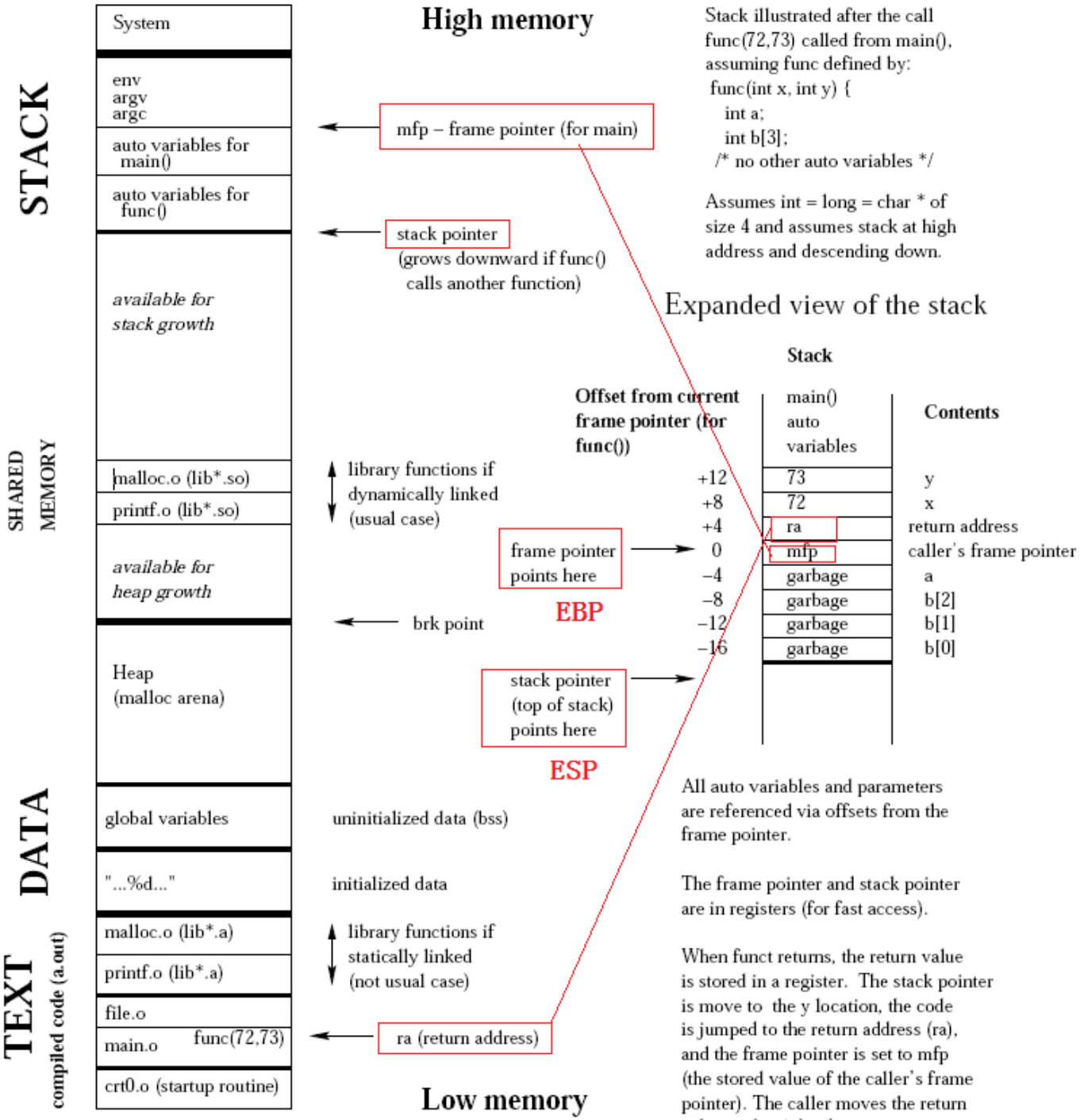

```
int x = 10;
int *i_ptr;
i_ptr = &x; // i_ptr -> the 1st byte of the 4 bytes long x.
cout << "x = " << x << endl;     // this prints x = 10
cout << "x = " << *i_ptr << endl; // this also prints x = 10


*i_ptr = 20;
cout << "x = " << x << endl;     // this prints x = 20
cout << "x = " << *i_ptr << endl; // this also prints x = 20

int y = 30;
i_ptr = &y;
cout << "y = " << *i_ptr << endl; // this prints y = 30
*i_ptr = 40
cout << "y = " << y << endl;     // this prints y = 40
```

Visualize


0x000000
int myint = 15
15
0xf3c453
0xf3c453
int* ptr = &myint
0xffffff


High memory
System
STACK
env
argv
argc
auto variables for main()
auto variables for func()
available for stack growth
SHARED MEMORY
malloc.o (lib*.so)
printf.o (lib*.so)
available for heap growth
Heap
(malloc arena)
DATA
global variables
"...%d..."
TEXT
compiled code (a.out)
malloc.o (lib*.a)
printf.o (lib*.a)
file.o
main.o func(72,73)
crt0.o (startup routine)
Low memory
mfp – frame pointer (for main)
stack pointer
(grows downward if func() calls another function)
library functions if dynamically linked (usual case)
brk point
uninitialized data (bss)
initialized data
library functions if statically linked (not usual case)
ra (return address)
Stack illustrated after the call func(72,73) called from main(), assuming func defined by:
func(int x, int y) {
int a;
int b[3];
/* no other auto variables */
Assumes int = long = char * of size 4 and assumes stack at high address and descending down.
Expanded view of the stack
Stack
Offset from current frame pointer (for func())
main() auto variables
Contents
+12 73 y
+8 72 x
+4 ra return address
0 mfp caller's frame pointer
−4 garbage a
−8 garbage b[2]
−12 garbage b[1]
−16 garbage b[0]
frame pointer points here
EBP
stack pointer (top of stack) points here
ESP
All auto variables and parameters are referenced via offsets from the frame pointer.
The frame pointer and stack pointer are in registers (for fast access).
When funct returns, the return value is stored in a register. The stack pointer is move to the y location, the code is jumped to the return address (ra), and the frame pointer is set to mfp (the stored value of the caller's frame pointer). The caller moves the return value to the right place.

Pointers and references


```
// pt.cpp: testing pointers
#include <iostream>

using namespace std;

void swap(int *a, int *b) { int temp=*a; *a=*b; *b=temp; }

int main() {
  int x = 1, y=9;
  swap(&x, &y);
  cout << "x = " << x << endl; // x = 9
  cout << "y = " << y << endl; // y = 1
  return 0;
}
```

Note that the pointers a,b are passed by value

Pointers to objects and the -> operator


```
void print_reversed_sentence(const string* s_ptr) {
  int start;
  int end = s_ptr->length()-1; // or end = (*s_ptr).length() - 1
  for (start = s_ptr->length()-1; start>=0; start--) {
    if ( ( (*s_ptr)[start] == ' ') && (start < end) ) {
      cout << s_ptr->substr(start+1,end-start) << ' ';
      end = start-1;
    }
  }

  if (start < end)
    cout << s_ptr->substr(start+1,end-start+1) << endl;
}
```

`(*obj_ptr).member` is the same as `obj->member`

Pointer to pointer, ad infinitum


```
#include <iostream>
using namespace std;

void swap(string **a, string **b)
{ string* temp = *a; *a = *b; *b = temp; }

int main() {
  string first("David"); string last("Blaine");
  string* p1 = &first; string* p2 = &last;

  swap(&p1, &p2);

  cout << "p1 points to " << *p1 << endl; // "Blaine"
  cout << "p2 points to " << *p2 << endl; // "David"
  return 0;
}
```

Arrays

ARRAY SIZE MUST BE A CONSTANT EXPRESSION

SOME EXTENSION ALLOWS "DYNAMIC" SIZE, NOT RECOMMENDED

ARRAY NAME CAN BE USED AS A POINTER TO THE FIRST ELEMENT OF THE ARRAY

Definition and initialization


```
int main() {
  const size_t s = 5;
  int A[s];
  int B[5] = {1, 2, 3, 4, 5};
  int C[] = {1, 2, 3, 4, 5}; // the same as saying int C[5] ...

  for (size_t i=0; i<s; i++) {
    A[i] = i*i;
    B[i] += A[i];
    C[i] += B[i];
  }

  for (size_t i=0; i<s; i++)
    cout << C[i] << '  ';

  cout << endl;
  return 0;
}
```

Extension – not recommended


```
int s = 5;
int A[s]; // should not be allowed, but OK with g++ 4.x
```

```
g++ -pedantic -ansi test_array.cpp

...
error: ISO C++ forbids variable-size array ‘A’
```

C-style strings


```
char name[] = "David";    // name has 6 elements, the last is implicitly '\0'
cout << sizeof(name) << endl; // prints 6

int i=0;
while (name[i] != '\0')
  cout << name[i++];
cout << endl;
cout << name << endl;

char name[5] = "David";    // compilation error
```

Initializing C-style string (char array)


```
#include <iostream>
using namespace std;

int main() {
  char name[]    = "David";
  char another[] = { 'D','a','v','i','d','\0' };

  cout << name << endl;
  cout << another << endl;

  return 0;
}
```

String literals

The expression `"David"` is called a string literal

```
char name[] = "This is a very long "
              "name and thus won't fit "
              "on a line";
cout << name; // get "This is a very long name and thus won't fit in a line"
```

Character arrays with '\0' in the middle

Perfectly fine to have them; just be careful!

```
char sa[] = "Only up to here\0The rest can still be printed";

cout << sa << endl;

for (i=0; i< sizeof(sa); i++) {
  if (sa[i] != '\0') cout << sa[i];
  else cout << "[NULL CHAR]";
}
cout << endl;

string str_obj = sa;
cout << str_obj;   // prints "Only up to here"
```

Why do we care about C-style strings?

- They are very efficient
- Necessary in system programming
- Some C++ functions take C-style string arguments

Command line arguments


```
int main()
int main(void)
int main(int argc, char **argv)
int main(int argc, char *argv[])
```

Multidimensional Arrays


```
const int m=2; const int n=3;
int A[m][n] = { {1,2,3},{4,5,6} };

// initialization must have bounds for all dimension, except the first
int B[][n] = { {10,20,30},{40,50,60} };
int C[m][n];
int i,j;

for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    C[i][j] = A[i][j] + B[i][j];

for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    cout << setw(2) << C[i][j] << '  ';
  }
  cout << endl;
}
```

Arrays and Pointers

- ARRAY NAME CAN BE USED AS POINTER
- WE CAN NAVIGATE ARRAYS USING POINTER ARITHMETIC
- POINTERS AND ARRAYS CAN BE USED INTERCHANGABLY IN ARGUMENT PASSING
- - AN ARRAY NAME CAN BE THOUGHT OF AS A CONSTANT POINTER

Array name as constant pointer


```
int a[5] = {1,2,3,4,5};
int* i_ptr = &a[0]; // i_ptr points to a[0]
i_ptr = a; // i_ptr points to a[0], equivalent to the above line
*i_ptr = 10; // now a[0] == 10
```

```
int a[5];
int* i_ptr;
i_ptr = a; // perfectly fine!
a = i_ptr; // compilation error!
```

Traversing array using pointer arithmetic


```
int A[5] = {1, 2, 3, 4, 5};
int i;
int* int_ptr;

// we can traverse A like this
for (i=0; i<5; i++)
  cout << A[i] << '  ';

// or like this
for (int_ptr=A; int_ptr != A+5; int_ptr++)
  cout << *int_ptr << '  ';
```

Pointer arithmetic


```
int a = 123;
int* int_ptr = &a;
cout << "+0:" << int_ptr << endl;     // +0:0x7fff6a9387c4
cout << "+1:" << int_ptr+1 << endl; // +1:0x7fff6a9387c8
cout << "+2:" << int_ptr+2 << endl; // +2:0x7fff6a9387cc
```

Pointer & array used interchangably in argument passing


```
void ps1(char* s) { while (*s != '\0') { cout << *s; s++; } }

void ps2(char s[]) { while (*s != '\0') { cout << *s; s++; } }

int main() {
    char* s1 = new char[7]; // create dynamically an array of 7 chars
    char s2[] = "abcde\n";
    int i=0;
    while (s2[i] != '\0') { s1[i] = s2[i]; i++; }

    ps1(s1); ps1(s2); // valid
    ps2(s1); ps2(s2); // also valid

    delete [] s1;
}
```