CS 33

Introduction to C
Part 2

Methods

?

- **C has functions**
- **Java has methods**
  - **methods implicitly refer to objects**
  - **C doesn't have objects**
- **Don't use the "M" word**
  - **it's just wrong**

```
for (;;)
  printf("C does not have methods!\n");
```

Swapping

Write a function to swap two ints

```
void swap(int i, int j) {


}
int main() {
    int a = 4;
    int b = 8;
    swap(a, b);
    printf("a:%d  b:%d", a, b);
}
```

Arguments are
passed by value

Swapping

Write a function to swap two ints

```
void swap(int i, int j) {
  int tmp;
  tmp = j; j = i; i = tmp;
}
int main() {
    int a = 4;
    int b = 8;
    swap(a, b);
    printf("a:%d  b:%d", a, b);
}
```

```
$ ./a.out
a:4  b:8
```

Darn!

This doesn't work because, when a function is called, copies are made of the arguments and it's these copies that are supplied to the function. Thus, if the function modifies its arguments, it's modifying only the copies. This is known as "pass by value".

Why "pass by value"?

- **Fortran, for example, passes arguments "by reference"**
- **Early implementations had the following problem (shown with C syntax):**

```c
int main() {
    function(2);
    printf("%d\n", 2);
}
void function(int x) {
    x = 3;
}
```

```
$ ./a.out
3
```

Note, this has been fixed in the (ancient) Fortran programming language (by recognizing that **literals** such as "2" are special). Since C passes arguments by value, this has never been a problem in C.

Variables and Memory

What does

    `int x;`

do?

- **It tells the compiler:**

  I want *x* to be the name of an area of memory that's big enough to hold an *int*.

What's memory?
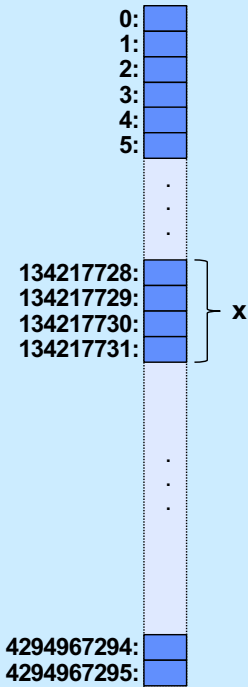
We'll discuss "what's an int" in a couple weeks.

Memory

- **"Real" memory**
  - **it's complicated**
  - **it involves electronics, semiconductors, physics, etc.**
  - **it's not terribly relevant at this point**
- **"Virtual" memory**
  - **the notion of memory as used by programs**
  - **it involves logical concepts**
  - **it's how you should think about memory (most of the time)**

Virtual Memory

- **It's a large array of bytes**
  - one byte is eight bits
  - an int is four consecutive bytes
  - so is a float
  - a char is one byte
- **The array index of a byte is its** *address*
  - the address of a larger item is the index of its first byte

virtual memory

```
0:
1:
2:
3:
4:
5:
.
.
.
134217728:  ⎫
134217729:  ⎬ x
134217730:  ⎪
134217731:  ⎭
.
.
.
4294967294:
4294967295:
```

In the diagram, x is an int occupying bytes 134217728, 134217729, 134217730, and 134217731. Its address is 134217728; its size is 4 (bytes).

Variables

- **Where**
  - **they refer to locations in memory**
- **Size**
  - **how much memory they refer to**
- **Interpretation**
  - **how to interpret the contents of memory**

- **All determined when they are declared**
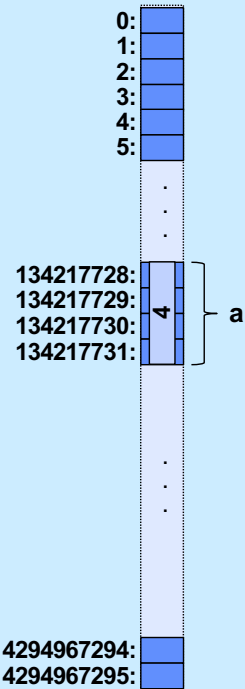- **None of the above change after declaration**

```
int x;   // sizeof(x) == 4
float y; // sizeof(y) == 4
char z;  // sizeof(z) == 1
```

Memory addresses in C

- **In C**
  - **you can get the memory address of any variable**
  - **just use the operator &**

```
int main() {
    int a = 4;
    printf("%p\n", &a);
}
```

```
$ ./a.out
134217728
```

Memory diagram (right side):

```
0:
1:
2:
3:
4:
5:
   .
   .
   .
134217728: ┐
134217729: │
134217730: │ 4  } a
134217731: ┘
   .
   .
   .
4294967294:
4294967295:
```

The "%p" format code in printf means to interpret the item being printed as being a pointer.

C Pointers

- **What is a C pointer?**
  - **a variable that holds an address**
- **Pointers in C are "typed" (remember the promises)**
  - **pointer to an int**
  - **pointer to a char**
  - **pointer to a float**
  - **pointer to <whatever you can define>**
- **C has a syntax to declare pointer types**
  - **things start to get complicated ...**

C Pointers

p is a pointer to an int

if you follow p, you find an int

```
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%p\n", p);
}
```
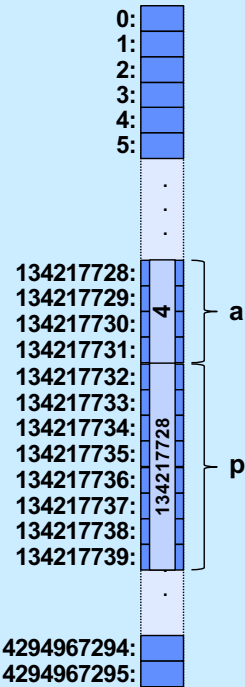
p is assigned the address of a

```
$ ./a.out
134217728
```

C Pointers

```c
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%p\n", p);
}
```

```
$ ./a.out
134217728
```

This slide assumes that pointers are 8 bytes long, which is the case for most computers we are likely to use as part of this class.

Some compilers might choose to order p in memory before a.

Note that both **a** and **p** are variables. Thus each is associated with a memory address, has a particular size, and has a particular interpretation. (Since an **int** can be positive, zero or negative, but an **address** can be only non-negative, their interpretations are slightly different. We'll explain this thoroughly in an upcoming lecture.)

C Pointers

- **Pointers are typed**
  - **the types of the items they point to are known**
  - **there is one exception (discussed later)**
- **Pointers are first-class citizens**
  - **they can be passed to functions**
  - **they can be stored in arrays and other data structures**
  - **they can be returned by functions**
- **Pointers have the properties of all variables**
  ```
  sizeof(int *) == sizeof(char *) == 8
  ```

The size of a pointer depends upon the type of computer. On most computers we use today, pointers have a size of 8 bytes. In some of our projects, we will use "32-bit mode" in which pointers are 4 bytes.

Swapping

What does this do?

```
void swap(int *i, int *j) {
   int *tmp;
   tmp = j; j = i; i = tmp;
}
int main() {
   int a = 4;
   int b = 8;
   swap(&a, &b);
   printf("a:%d  b:%d\n", a, b);
}
```
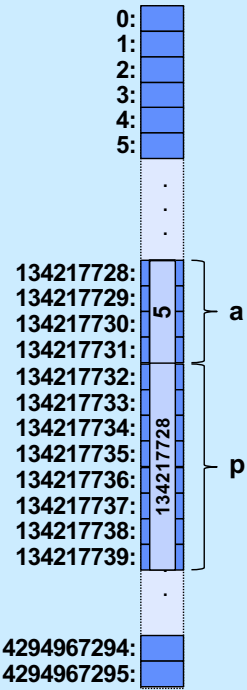
```
$ ./a.out
a:4  b:8
```

Damn!

C Pointers

- **Dereferencing pointers**
  - **accessing/modifying the value pointed to by a pointer**

```
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%d\n", *p);
    *p = *p + 1;
    printf("%d\n", *p);
}
```

```
$ ./a.out
4
5
```

Memory addresses:
```
0:
1:
2:
3:
4:
5:
   .
   .
   .
134217728:
134217729:
134217730:      5        } a
134217731:
134217732:
134217733:
134217734:
134217735:  134217728
134217736:               } p
134217737:
134217738:
134217739:
   .
4294967294:
4294967295:
```

Keep in mind that though we are changing p's value (by assigning to it the address of a), we are not changing the memory address associated with p – it remains 134217732 (and its size remains 8).

Dereferencing C Pointers

```
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%d\n", *p);
    *p = *p + 1;
    *p += 3;
    printf("%d\n", a);
}
```

```
$ ./a.out
4
8
```

Note that "*p" and "a" refer to the same thing after p is assigned the address of a.

"x += y" means the same as "x = x+y". Similarly, there are −=, *=, and /= operators.

Swapping

```
void swap(int *i, int *j) {
   int tmp;
   tmp = *j; *j = *i; *i = tmp;
}
int main() {
   int a = 4;
   int b = 8;
   swap(&a, &b);
   printf("a:%d  b:%d\n", a, b);
}
```

Hooray!

```
$ ./a.out
a:8  b:4
```
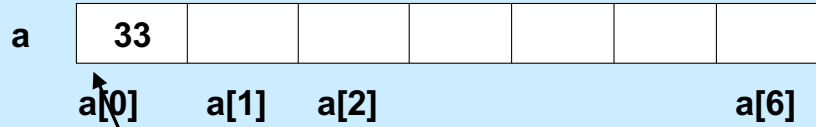
Quiz 1

```
int doubleit(int *p) {
    *p = 2*(*p);
    return *p;
}
int main() {
    int a = 4;
    int b;
    b = doubleit(&a);
    printf("%d\n", a*b);
}
```

What's printed?

a) 8
b) 16
c) 32
d) 64

This quiz makes up a (very small) part of your grade.

Pointers and Arrays

a

33						

a[0]   **a[1]**   **a[2]**   **a[6]**

p
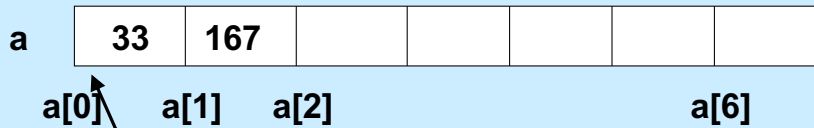
```
int main() {
    int a[7];
    int *p;
    p = &a[0];
    *p = 33;
}
```

The pointer p points to the first element of the array a. Thus a[0] and *p have identical values.

Pointer Arithmetic

Pointers can be incremented/decremented
  – what this does depends on its type

a	33	167					

a[0]   a[1]   a[2]   a[6]

p

```
int main() {
    int a[7];
    int *p;
    p = &a[0];
    *p = 33;
    *(p+1) = 167;
}
```

Adding one to a pointer, rather than increasing its value by one, causes it to refer to the next element. Thus, if the size of what it refers to is 4 (which is the case for pointers to ints), adding one to the pointer increases its value by 4 (thus making it point to the next 4-byte value).

Pointer Arithmetic

Pointers can be incremented/decremented
  – **what this does depends on its type**

a
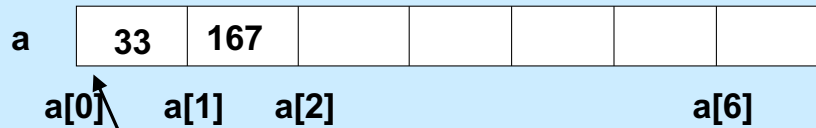
a[0] **a[1]** **a[2]** **a[6]**

p
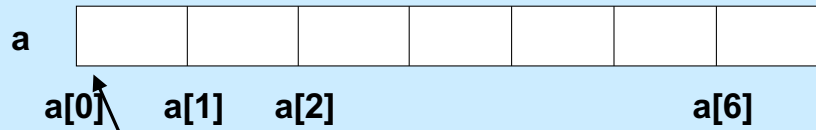
```
int main() {
    int a[7];
    int *p;
    p = &a[0];
}
```

Now `p` and `a`
have the
same value

Note that setting **p** equal to the address of the first element of the array a is equivalent to setting **p** to the value of **a**.

Pointer Arithmetic

Pointers can be incremented/decremented
   – what this does depends on its type

a | 33 | 167 | | | | | |

a[0]   **a[1]**   **a[2]**   **a[6]**

p | |

```
int main() {
    int a[7];
    int *p;
    p = a;
    *p = 33;
    p[1] = 167;
}
```

The array name represents a pointer to its first element

A pointer to the first element of an array can be used as if it were the array itself. Thus, in this example, there's little difference between how one uses "p" and "a".

An array's name represents a pointer to its first element.

Pointers and Arrays

`p = &a[0];`   **can also be written as**   `p = a;`

`a[i];`   **really is**   `*(a+i)`

- **This makes sense, yet is weird ...**
  - `p` **is of type** `int *`
    - **it can be assigned to**
      `int *q;`
      `p = q;`
  - `a` **sort of behaves like an** `int *`
    - **but it can't be assigned to in the same way**
      ~~`a = q;`~~

The name of a local array represents a pointer to its first element. But we treat this pointer as a constant – it can't be modified (if we did so, what would we do with the storage that it previously pointed to?)

Note that we can assign to array arguments of functions – we explain this soon.

Non-Array Variables

- **int** i
  - **four bytes of memory are allocated for** i
    - `sizeof(i) == 4`
  - i **represents the contents of this memory, interpreted as an int**
  - **it makes sense to do, for example**
    - `i = 7; // changes the contents of i`
- **int** *p
  - **8 bytes of memory are allocated for p**
    - `sizeof(p) == 8`
  - p **represents the contents of this memory, interpreted as an int ***
  - **it makes sense to do, for example**
    - `p = &i; // changes the contents of p`

All non-array variables, whether pointer or not, are treated the same way: the variable name represents the contents of the memory associated with that variable.

Array Variables

- **int** A[6]
  - **24 bytes of memory are allocated for** A
    - `sizeof(A) == 24`
  - A **represents the address of the first byte**
  - *A **is the value of the first int (as if** A **were an int *)**
  - **it does not make sense to do**
    - `A = &i; // would change the location of A`
- **int** *p = A;
  - **8 bytes of memory are allocated for** p
    - `sizeof(p) == 8`
  - p **represents the contents of this memory**
  - *p **is the same as** A[0]
  - **it makes sense to do, for example**
    - `p = &i;`

Array variables are different from other variables. In particular, the name of an array variable is associated with the address of the array, while the name of all other kinds of variables are associated with the contents of the memory associated with that variable. Furthermore, if we dereference an array variable, the result is not the contents of the entire array, but just the contents of its first element.

Arrays and Functions

initialized with a copy of the argument

```
int func(int *a, int nelements) {
    int i;
    for (i=0; i<nelements; i++) {
        *(a+i) = i;
    }
    return nelements;
}

int main() {
    int array[1000000000] = ... ;
    printf("result = %d\n", func(array, 1000000000));
    return 0;
}
```

When an array variable is used in a function call, it continues to represent the address of the array. Thus the called function is passed an address to the array, not the entire array. Among the reasons for this design decision in C is that it allows large arrays to be passed without having to copy all their elements. Note that since only the pointer to the first element is passed, we need to supply the length of the array if we need to know what it is (such as here where we are initializing its contents).

Equivalently

initialized with a copy of the argument

```
int func(int a[], int nelements) {
    int i;
    for (i=0; i<nelements; i++) {
        a[i] = i;
    }
    return nelements;
}

int main() {
    int array[1000000000] = ... ;
    printf("result = %d\n", func(array, 1000000000));
    return 0;
}
```

II–28

We can use this syntax, which has exactly the same effect as that of the previous slide, but makes it clear (to the human reader) that **a** is really an array.

Equivalently

ignored

```
int func(int a[500], int nelements) {
    int i;
    for (i=0; i<nelements; i++) {
        a[i] = i;
    }
    return nelements;
}

int main() {
    int array[1000000000] = ... ;
    printf("result = %d\n", func(array, 1000000000));
    return 0;
}
```

Here we've added a size for the array in the function arguments. But, though this is syntactically correct (gcc will not complain), the size is ignored. What's passed to **func** is just the pointer to the original array. Only with the **nelements** argument will **func** know how large the array is.

Parameter passing

Passing arrays to a function

```c
int average(int a[], int size) {
    int i; int sum;
    for(i=0,sum=0; i<size; i++)
        sum += a[i];
    return sum/size;
}
int main() {
    int a[100];
    …
    printf("%d\n",average(a,100));
}
```

Here is another example of passing an array to a function. We need to pass the size of the array as well, assuming the function needs to know the array's size.

Swapping

Write a function to swap two entries of an array

```
void swap(int a[], int i, int j) {
    int tmp;
    tmp = a[j];
    a[j] = a[i];
    a[i] = tmp;
}
```

Selection Sort

```c
void selectsort(int array[], int length){
  int i, j, min;
  for (i = 0; i < length; ++i){
    /* find the index of the smallest item from i onward */
    min = i;
    for (j = i; j < length; ++j) {
      if (array[j] < array[min])
        min = j;
    }
    /* swap the smallest item with the i-th item */
    swap(array, i, min);
  }
  /* at the end of each iteration, the first i slots have the i
     smallest items */
}
```

Note that C uses the same syntax as Java does for conditional (if) statements. In addition to relational operators such as "==", "!=", "<", ">", "<=", and ">=", there are the conditional operators "&&" and "||" ("logical and" and "logical or", respectively).

Arrays and Arguments

```c
int func(int arg[]) {
    int array2[6] = {4, 5, 6, 7, 8, 9};
    arg[1] = 0;
    arg = array2;
    return arg[3];
}

int main() {
    int array1[4] = {0, 1, 2, 3};
    int x = func(array1);
    printf("%d, %d\n", x, array1[1]);
    return 0;
}
```

array1 → 0
0
2
3

x: 7

arg
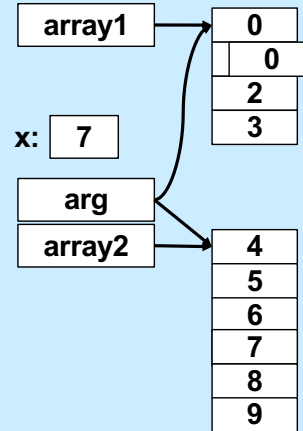
array2 → 4
5
6
7
8
9

```
$ ./a.out
7 0
```

In this example, we've declared **array1** and **array2** in **main** and **func**. Both declarations allocate storage for arrays of **int**s. Both **array1** and **array2** refer (by pointing to the first elements) to the storage allocated for the arrays. What memory locations they refer to can't be changed (though the contents of these locations can be changed).

In the definition of **func**, **arg** is an argument that acts as a variable that's initialized with whatever is passed to **func**. In the slide, **func** is called with **array1** as the argument. Thus, **arg** is initialized with **array1**, which means it's initialized with a pointer to the first element of the array referred to by **array1**. But this initial value of **arg** is not permanent -- we're free to change it, as we do when we assign **array2** to **arg**.
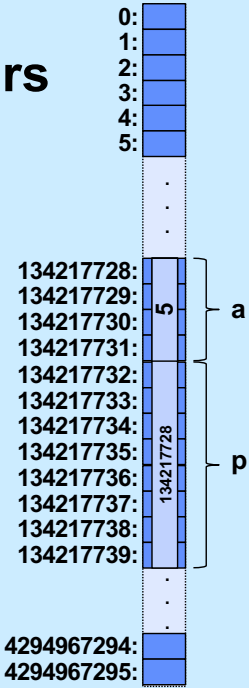
Arrays and Arguments

```
void func(int arg[]) {
   /* arg points to the caller's array */
   int local[7];    /* seven ints */
   arg++;           /* legal */
   arg = local;     /* legal */
   local++;         /* illegal */
   local = arg;     /* illegal */
}
```

Dereferencing C Pointers

```
int main() {
    int *p; int a = 4;
    p = &a;
    (*p)++;
    printf("%d %p\n", *p, p);
}
```
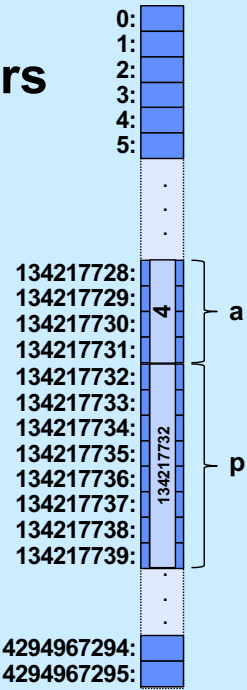
```
$ ./a.out
5 134217728
```

0:
1:
2:
3:
4:
5:
.
.
.
134217728:
134217729:   5   } a
134217730:
134217731:
134217732:
134217733:
134217734:
134217735:   134217728   } p
134217736:
134217737:
134217738:
134217739:
.
.
.
4294967294:
4294967295:

Dereferencing C Pointers

```
int main() {
    int *p; int a = 4;
    p = &a;
    *p++;
    printf("%d %p\n", *p, p);
}
```

```
$ ./a.out
134217732 134217732
```

0:	
1:	
2:	
3:	
4:	
5:	
.	
.	
.	
134217728:	
134217729:	4 ⎤ a
134217730:	
134217731:	⎦
134217732:	⎤
134217733:	
134217734:	
134217735:	134217732 ⎬ p
134217736:	
134217737:	
134217738:	
134217739:	⎦
.	
.	
.	
4294967294:	
4294967295:	

Operator precedence is hard to remember! ("++" takes precedence over "*".)

Note that even though ***p** is an int, but it's printed as an 8-byte pointer, what's printed is its value as an int. Exactly why this is so (and why it could be a problem) is something we'll discuss in a week or two.

Dereferencing C Pointers

```c
int main() {
    int *p; int a = 4;
    p = &a;
    ++*p;
    printf("%d %p\n", *p, p);
}
```

```
$ ./a.out
5 134217728
```

Here it's clear that the * operator is applied before the ++ operator.

Quiz 2

```
int func(int arg[]) {
    arg++;
    return arg[0];
}

int main() {
    int A[3]={10, 11, 12};
    printf("%d\n",
       func(A));
}
```

What's printed?

a) 9
b) 10
c) 11
d) 12

Quiz 3

```
int func(int a[]) {
   int b[5] = {10, 11, 12, 13, 14};
   a = b;
   return a[1];
}

int main() {
   int array[50];
   array[1] = 0;
   printf("result = %d\n",
      func(array));
   return 0;
}
```

This program prints:

a) 0
b) 10
c) 11
d) nothing: it doesn't compile because of a syntax error

Note how we initialize the contents of array **b** in **func**.

Quiz 4

```c
int func(int a[]) {
   int b[5] = {10, 11, 12, 13, 14};
   a = b;
   return a[1];
}

int main() {
   int array[5] = {9, 8, 7, 6, 5};
   func(array);
   printf("%d\n", array[1]);
   return 0;
}
```

This program prints:

a) 7
b) 8
c) 10
d) 11

The Preprocessor

`#include`

- **calls the preprocessor to include a file**

What do you include?

- **your own *header* file:**
  `#include "fact.h"`
  - **look in the current directory**
- **standard *header* file:**
  `#include <assert.h>`
  `#include <stdio.h>` — **Contains declaration of *printf* (and other things)**
    - **look in a standard place**

The preprocessor modifies the source code before the code is compiled. Thus, its output is what is passed to gcc's compiler.

Note that one must include **stdio.h** if using **printf** (as well as some other functions) in a program.

On most Unix systems (including Linux, but not OS X), the standard place for header files is the directory /usr/include.

Function Declarations

fact.h

main.c

```
float fact(int i);
```

```
#include "fact.h"
int main() {
    printf("%f\n", fact(5));
    return 0;
}
```

It's convenient to package the declaration of functions (and other useful stuff) in header files, such as **fact.h**, so the programmer need simply to include them, rather than reproduce their contents.

The source code for the **fact** function would be in some other file, perhaps as part of a library (a concept we discuss later).

#define

```
#define SIZE 100
int main() {
    int i;
    int a[SIZE];
}
```

#define
- **defines a substitution**
- **applied to the program by the preprocessor**

#define

```
#define forever for(;;)
int main() {
    int i;
    forever {
      printf("hello world\n");
    }
}
```

The #define directive can be used for pretty much anything, such as segments of code as shown in the slide. (It's not its concern as to whether the code segments are useful!)

assert

```
#include <assert.h>
float fact(int i) {
   int k, res;
   assert(i >= 0);
   for(res=1,k=1; k<=i; k++)
     res = res * k;
   return res;
}
int main() {
   printf("%f\n", fact(-1));
```

assert
- **verify that the assertion holds**
- **abort if not**

```
$ ./fact
main.c:4: failed assertion 'i >= 0'
Abort
```

The assert statement is actually implemented as a macro (using #define). One can "turn off" asserts by defining (using #define) NDEBUG. For example,

#include <assert.h>

…

#define NDEBUG

…

assert(i>=0);

In this case, the assert will not be executed, since NDEBUG is defined. Note that one also can define items such as NDEBUG on the command line for gcc using the –D flag. For example,

gcc –o prog prog.c –DNDEBUG

Has the same effect as having "#define NDEBUG" as the first line of prog.c.